Sierra Wireless Open AT® TUTORIAL

Develop Open AT® Applications
for M2M & Automotive Solutions

OpenAT

SIERRA WIRELESS™ | AirPrime™

PREFACE

Welcome to the programming guide to Open AT, the operating system dedicated to wireless Machine-to-Machine (M2M) development.

About M2M

Simply said, M2M includes all businesses, technologies and services whose primary purpose is to create a data link exchange between machines, and not between individuals. Typical examples are remotely reading and managing power consumption (metering), in-vehicle diagnostic, assistance and emergency systems (fleet and automotive), remote patient monitoring (healthcare), home & industrial alarms (security) and many more.

The objective of Open AT is to provide a complete software and tool suite to facilitate the creation, deployment and management of such applications in small and constrained embedded environments.

The history of Open AT

Open AT was created in 1998, a couple of years after Wavecom (now Sierra Wireless) created the first cellular wireless module, named WISMO 1A. At that time, it was a 40mm by 64mm, dual-sided 900 MHz GSM device, and was running on an ARM6 processor. It was an impressive industrial achievement at the time, even if its features pale in comparison to the todayr's WISMO 228: as small as 25mm by 25mm, single sided, GSM/GPRS, 4 bands, with powerful ARM926 processor running at hundreds of MHz.

In these early days of GSM technology, we were selling most of these devices to handset manufacturers. But these small modules were already running much more than a proprietary Protocol Stack as the whole User Interface was also sharing the modest ARM5 processor. Wavecom was providing a simple SDK to create a sophisticated UI, although at this time the concept of color screen and multimedia graphical devices was not very common.

Alongside with the handset business, Wavecom was an early believer in M2M and the very same WISMO 1A device had a specific software version that would enable it to be driven through AT commands, similar to a regular PSTN modem. GPRS did not exist at the time, but CSD (Circuit-Switched Data), Fax and SMS were used for M2M.

The usual architecture was to have a micro-controller or processor running a user application (early Telemetry, Fleet management or even Wireless Local Loop) that would control the WISMO 1A device through these AT commands.

Of course, a natural parallel with the handset architecture model was immediately to allow the same capability to run the application software directly on the module instead of an external processor, as in many cases an M2M application was less demanding than a handset UI. But it was not a simple development cycle and the flexibility requirements were (and still are) very different:

- A handset UI is often a bigger effort, hence it made sense for customers to allocate a fair amount of time and efforts to understand a SDK – it is worth the investment, even if the SDK is complex
- Change of features, updates, improvement happen quite often in M2M, so a rapid design cycle for small application was needed where, to the contrary, UI changes after a handset commercial launch are rare
- Handset suppliers can afford the burden of doing an extensive certification process thanks to the high volume and the small amount of software version per model. M2M customers cannot.

So we then started to consider a programming paradigm that would be quick to understand, tailored for small applications and would not require wireless certification (i.e. to link the application software with the embedded protocol stack). The first idea was to use scripting or interpreted languages, from Basic to interpreted C but also Java and Javascript or Python. This was quickly proven to not be the right solution simply because of certain constraints for efficient embedded software:

- Embedded developers are very familiar with C and they like it, despite (or because of) its very “unsafe” nature
- The interest of using our module as an application processor is a mix of cost, size and availability of SDK (pre-existing libraries). But any sort of interpreted language has a large memory footprint that would at least partly void the cost interest
- A lot of applications, even if they are small, do require some sort of real time capability, to drive IO or peripherals, so at least one language “close to the metal” was needed.

Finally, using C as the main programming language was the best compromise, with a decent isolation and protection between user space and kernel space. Thus the module was delivered with pre-loaded firmware and the possibility to download a user application without linking both.

Another consideration taken into account was the possibility to provide an easy transition from using an external processor connected to a modem through AT commands to sharing the same CPU. Obviously, allowing the user application to send the same AT commands, but directly from within the module provided this smooth transition - “Open AT” was born.

Many years and millions of devices later, Open AT’s simple AT command mode has been improved with a complete API to facilitate application development. This is the ‘adl’ interface which allows application to do native calls to the kernel, instead of sending and receiving AT commands. This is faster, easier, organized by family of services and a good basis to add extensions, or plug-ins (like GPS, TCP/IP, Security and more). Nevertheless, the built-in capability to send and receive AT commands has been and will be maintained.

What might be surprising while discovering this system is the choice of an event-driven programming paradigm, which is not so usual in the embedded space. Although it requires a slightly different way to build a program, this brings multiple benefits, including:

- Adapted to sensor-driven environment
- Easy way to overload AT commands to create user-specific versions
- Paves the way for strong RAD (Rapid Development) tools that bring a “modeling by object” approach

Tools are also a very important topic that all developers pay great attention to - in fact, many surveys show that this is a primary concern. A few years ago, we decided to base our entire IDE on the well-recognized, open Eclipse™ suite. The free Developer Studio now includes all modern features of a fully integrated environment and there is continued focus to deliver a very polished development and debugging experience.

One last word regarding programming paradigm and languages: the initial development language chosen for Open AT is C, for the benefit of speed (real time) and efficiency. But it is also understood that different needs require different languages. Instead of opposing C and scripting, both are provided to be able to write the low layer, HW-connected drivers in C for data extraction and be able to benefit from the ease and speed of development of 4G languages for data manipulation.

Open AT does not impose only one language and Lua has been added as an open source, free to use Language and Virtual Machine. It also offers very easy integration with what is written in C, all with the minimum penalty for a VM and a 4G language. To take full benefit of such a system, a complete 4G modeling environment named Developer Suite is available and we plan to bring more on languages and tools soon.

Sierra Wireless hopes this guide will serve well in discovering and using Open AT. Don't hesitate to join our forum and exchange with developers at https://forum.sierrawireless.com/.

Philippe Guillemette
Chief Technology Officer

Contents

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1

About this Training Courseware

1. Objective

This book will guide you through the working of Open AT with detailed step-by-step instructions. This book is also the support of the Open AT training organized regularly by Sierra Wireless allowing developers discover easily the word of Open AT.

The following features of this document will guide you through the installation, setup and working of Open AT:

- Clear examples to understand.
- Images to help you get the picture
- Exercises to help you gauge your understanding of concepts and instructions
- Code samples that you can use to create Open AT Application.
- Notes, important tips, points to remember and summaries to help you recall the salient points of each chapter
- A glossary that will give you explanations to technical terms in this document

If you need more information, please contact your Sierra Wireless distributor.

2. Audience

This training guide is addressed to all developers want to start M2M design application with Open AT.

3. Pre-requisites

These pre-requisites must be in place before starting Open AT training or before developing your first Open AT application:

- Sierra Wireless Development kit along with required power supplies and adaptors

- SIM cards with the following features enabled according to your usage:
  - Basic GSM features such as SMS and voice call
  - PIN1, PIN2, PUK1 and PUK2 codes
  - GPRS (if required)
  - GSM data call (if required)

  - GSM fax call (if required)
  - WAP (if required)
  - CLIP (Caller Line Identification Presentation) (if required)
  - Call waiting, call barring, call forwarding features (if required)

- Hardware Requirements: The main requirements for the PC to support the Developer Studio are:
  - Pentium 300MHz (or higher) processor with at least 128 MB of RAM and 500 MB free hard disk space.
  - A CD ROM drive (for software installation)
  - At least one free COM (serial) port for the communication with the target product.

- Software Requirements: Following is the list of software which is required to build the Open AT Application:
  - Developer Studio (including GCC compiler)
  - ARM compiler (if required)

- System Requirements: To be able to install the Developer Studio, your system must support one of the following operating systems:
  - WINDOWS 98
  - WINDOWS NT
  - WINDOWS Me
  - WINDOWS 2000
  - WINDOWS XP
  - WINDOWS 7

- Sierra Wireless Software Suite from an USB Key or downloaded from www.sierrawireless.com

- Working experience of AT commands and C language

NOTE:
The GCC compiler provided within the Developer Studio installation setup can run only on Windows NT/2000/XP systems. It is not supported on Windows 98/ME systems.

CHAPTER 2

Introduction to Open AT®

1. Objective

The objective of this chapter is to introduce you the Open AT concept and the different modes to use it.

2. What is Open AT?

Open AT is the way to embed and execute your software application into the Sierra Wireless embedded module. Then your embedded module becomes your development platform able to host your M2M application.

The following image depicts a microcontroller-based application without Open AT Application. In such a situation, all the applications are external to the embedded module.

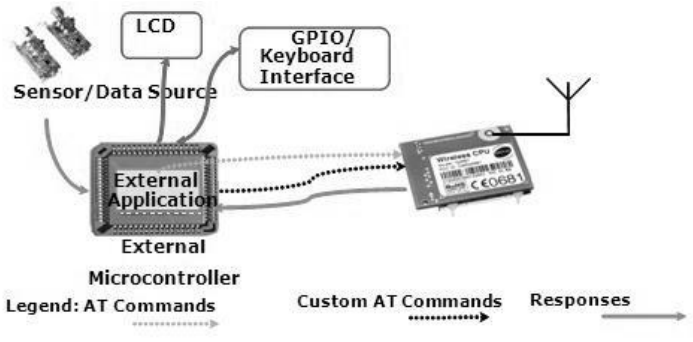


Figure 1 –Without Open AT Application

In such a scenario, without Open AT Application, you must have interfaces for the hardware, and also develop custom applications that will be embedded in the external micro-controller to interact with embedded module and with the external devices (by GPIO or BUS interface).

With Open AT Application, the external micro-controller is not required, and therefore the Open AT Application takes control of the embedded module and interacts with the external devices. This is depicted in the following image.

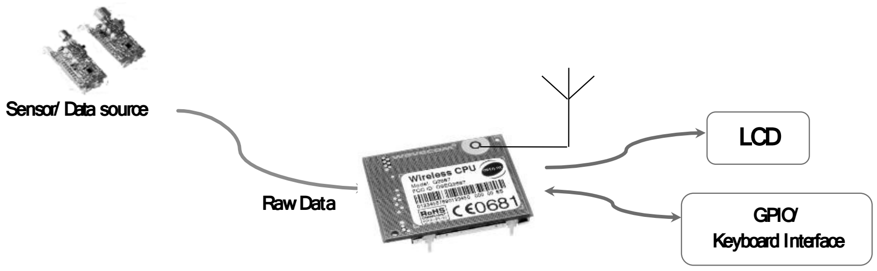


Figure 2 - With Open AT Application

Open AT Application allows you more flexibility. Open AT Application allows you to modify the response to basic AT commands and therefore customize according to the functionality required. In addition to the basic AT commands, you can also develop custom commands and add to the available command set. The required functionality can be implemented in these commands.

This provides more power to the wireless developer and eases the programming task. The following figure depicts this advantage with a scenario in which custom commands are used.

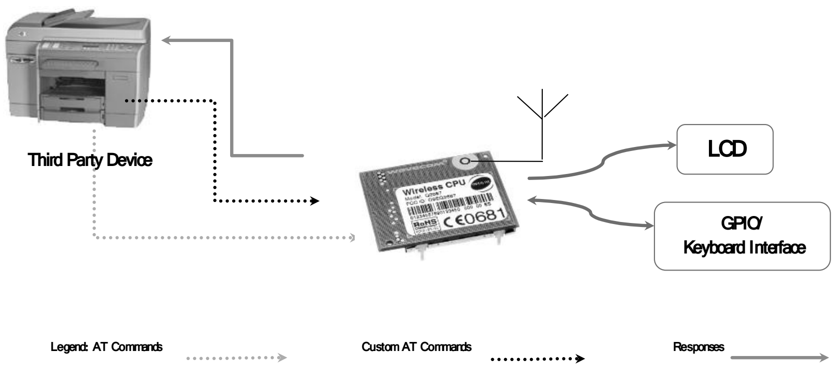


Figure 3 - Open AT Application with Custom Commands

3. Areas of Application

Open AT OS can provide functionality in a wide variety of applications, such as:

- Systems that use standard GSM, GPRS and EDGE services for status reporting
- Systems that provide internet connectivity and access to Web services like FTP, HTTP, POP3, SMTP and PING
- Systems that act as TCP/UDP client/server machines and provide services to other nodes
- Systems that provide location information in automotive systems using GPS (Global Positioning System) services
- Applications that send sensor information (read through GPIOs) to remote base locations
- Applications that store sensor data or other vital information in flash (ROM) and help protect critical information for state of the art information processing systems
- Applications that provide WAP-related services

Domains where Open AT Applications can be used include Automotive, Fleet Management, Remote Asset Monitoring, Telemetry and Telematics etc.

The Application figure depicts an example of Open AT usage. In case of a motor accident, the sensors fitted in the automobile send the information to the base server, which in turn informs the emergency helpline, prompting it to send help to the accident site. The Open AT Application running on the handles this process of collecting and sending the information to the base server.

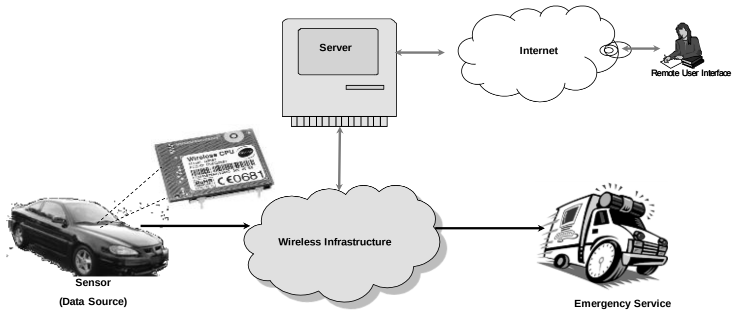


Figure 4 - Open AT Application for Automotive

4. Advantages of using Open AT

The advantages of using Open AT:

The Open AT helps reducing the development and testing time considerably. You can concentrate on the functionalities of the application instead of the lower level technical details.

Open AT reduces the cost of the product. Without Open AT, a considerable amount of time is spent interfacing hardware and creating drivers which are used to provide external stimuli (information) to the embedded module. But Open AT OS provides APIs with which the power of the embedded module can be easily manipulated without any external interfacing hardware or software.

Open AT Application is tightly integrated with the Firmware. As the Open AT OS (on which the application is built) is extensively tested to work with the Firmware, the level of integration and reliability of the Open AT Application meets user expectations. With Open AT OS (library) the Open AT Application can interact efficiently with the Firmware.

Open AT Application allows you more flexibility; in addition to the basic AT commands, you can also develop custom commands which can add to the available command set. The required functionality can be provided in these commands. Hence, the overall flexibility of programming can be increased.

Open AT can be used to create intelligent internet-enabled applications directly on the embedded module. This shortens development time and reduce materials cost.

This Operating System allows defining multiple tasks which can run parallel. It provides RTOS features which:

- Allows writing time critical applications along with GSM/GPRS/EDGE capabilities.
- Guarantees task execution within a specified time constraint.
- Provides access to high granularity timers (<1ms).

- Allows choosing between different scheduling algorithms (Priority or time based).
- Provides access to various interrupts such as Bus, UART, ADC, Audio, Timer, Pins.
- Allows configuring processor clock speed.

Summary

The following points have been covered in this chapter:

- Without Open AT Application, all the applications are external to the embedded module. With Open AT Application, the Open AT Application takes control of the embedded module, and interacts with the external devices.

CHAPTER 3

Embedded Software Architecture

1. Objective

This chapter describes the module embedded software components and architecture. It includes the minimum to know about the Open AT OS: main ADL interfaces, memory resources and running modes of Open AT Application.

2. Embedded Software Architecture

Open AT Embedded Software Suite (OASiS) is a set of embedded module software components for Open AT. These components are provided by Sierra Wireless. It includes:

- A **Firmware** which is the embedded software core provided by Sierra Wireless. It provides the wireless connectivity services and the AT parser services. It natively includes the software agents to manage remotely devices through the Sierra Wireless AirVantage® Management Service portal.
- An **Operating System**, named Open AT OS. It provides an execution environment to host your application into the Sierra Wireless embedded module. This component is delivered as a library
- A large set of **Plug-ins for Open AT**, which are providing supplementary services which are either purely software such as internet plug-in or associated to hardware expansion such as the location plug-in for GPS services

The Open AT OS APIs interacts directly with the Firmware. The Open AT Application is built on top of these APIs. The Firmware is present in the embedded module as one binary file. The Open AT Application also compiles to a binary file which lies above the Firmware. The software thus follows a dual binary architecture.

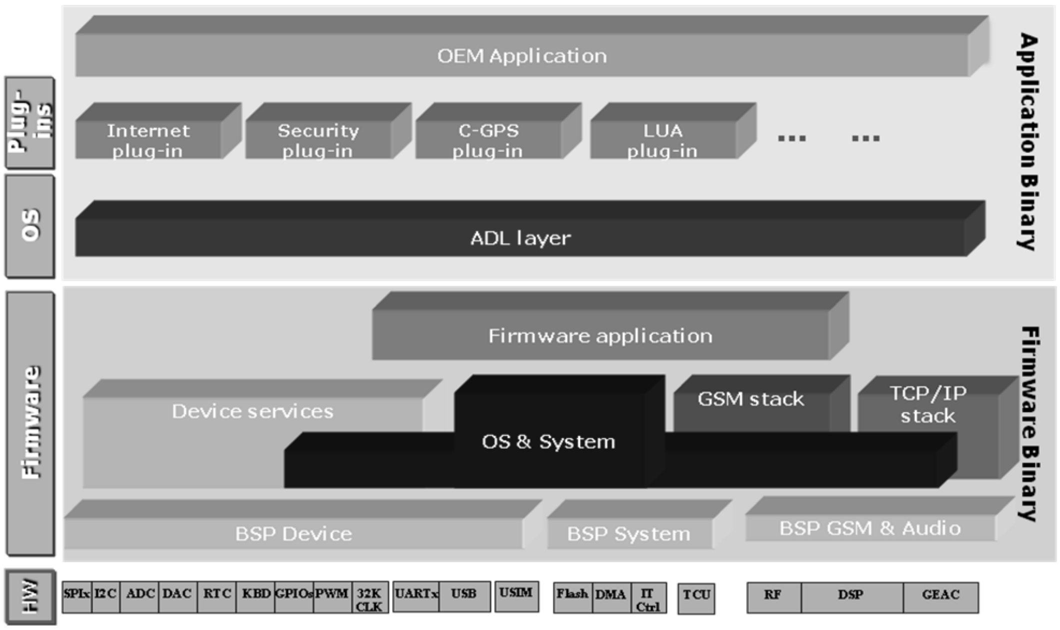


Figure 5 – Embedded Software Architecture

The Open AT Application binary file is built by the Open AT developer. It is the Open AT developer source code compiled and linked with the Open AT OS library and optionally with other plug-in libraries. This binary is downloaded into the flash memory and runs concurrently with the Firmware, using the Open AT OS interface and managed by the same Real Time OS. (For more details, refer to the chapter 13 flash memory and chapter 30 RTOS) In this way, self-reliant applications can be created and programmed in the flash memory to run without any user intervention.

3. ADL Interfaces

The Open AT OS is a multitasking Operating System. This mode has the following features:

- High-level event processing interface
- Highly modular capabilities
- Event processing using user-defined call-back functions
- Easily maintainable code
- High ease of programming
- Multitasking which allows different tasks to run parallel
- Tasks can have different configurable priorities which is very useful to decide the real-time nature of the system

The Open AT OS is providing an APIs named ADL which includes:

- AT command APIs: Software interfaces that provide access to the Firmware functions.
- OS API: Software interfaces that provide access to operating system functions.

- FCM API: Software interfaces that provide access to the Flow Control Manager functions (secure access to V24 and Data IO flows). For more details, refer to the chapter on the flow control manager.
- GPIO API: Software interfaces that provide access to the GPIO services.
- GPRS/EDGE API: Software interfaces that provide access to the GPRS/EDGE services (for authentication and IPCP information).
- BUS API: Software interfaces that provide control on bus devices (as SPI, Parallel or I2C bus).
- List API: A set of list-processing functions.
- DOTA I/II/III API: A set of APIs to enable Download Over The Air functionality.
- Sound API: Software interfaces to control the playing of regular and DTMF tones.
- Standard API: A standard set of C functions, in addition to some string processing functions.
- RTOS API: A set of APIs that supports RTOS feature.
- Registry API: A set of APIs that provides access to the platform registry.
- Error Management API: A set of APIs for managing errors.
- Secured Data API: A set of APIs for storing secured data.

4. Mandatory Embedded Application Code

To develop an Open AT Application certain mandatory APIs must be implemented. The following sections describe the mandatory ADL APIs.

To define the Open AT Application call stack size follow the prototype below:

```
const u16 wm_apmCustomStackSize = 1024;
```

Before Open AT OS 6.30, the entry point function that is called during the Open AT Application initialization was:

```
void adl_main ( adl_InitType_e   InitType )
{
}
```

This entry point has been kept for **backward compatibility**. However currently Open AT OS allows to define multiple tasks. Each task is defined 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

Each task is defined using the following structure in Open AT Application.

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

Please refer the RTOS section in this book of the course material for more details on this feature.

5. Open AT OS Memory Resources

Based on the memory type of the embedded module, different sizes of Open AT Applications can be downloaded in the embedded module. The following table lists the memory available for Open AT:

Combo memory (flash/ram)	Open AT non-volatile (code, A&D)	Open AT RAM	Open AT Object
64/16	5 MBytes	1 MByte	384 Kbytes
32/8	1,5 Mbytes	256 Kbytes	128 KBytes

The global variable, call stack and dynamic memory allocated are all part of the RAM allocated to the Open AT Application. The RAM size used by the call stack is dependent on the Open AT Application. The call stack uses at least 1KB of RAM. In addition to that, 0.5KB of memory is used by internal variables of the library. Hence in case an Open AT Application is developed on an embedded module with 128kbytes of RAM available to Open AT, then the RAM memory is shared between Open AT Application and Open AT OS variables. The RAM memory will have

- Open AT Application/Task(s) stack
- Open AT OS and Open AT Application global variables
- Open AT OS and Open AT Application heap

The Firmware and Open AT Application manages their own RAM area and any access from one of these programs into the other will cause the embedded module to reboot. In case of illegal access of RAM, the Trace View screen displays the message “ARM exception: 1 xxx” where xxx is the address of the location that the Open AT Application was trying to access.

In case an Open AT Application uses more than the specified RAM or ROM area, then the behavior of the program becomes erratic.

6. Open AT Application Running Modes

There are two execution modes for any Open AT Application. The following section describes the target and remote modes of execution.

6.1. Target Mode

The Open AT Application is downloaded in the embedded module (programmed in flash) and executed in the target mode. The Open AT Application uses the resources of the embedded module to perform the required operation. This mode is used when a standalone Open AT Application must be deployed in the embedded module. The application is self-reliant and can run without user intervention.

6.2. Remote Mode

In the remote mode, the Open AT Application is executed on the development environment (PC) and uses its own resources like memory and clock. This mode provides the facility of line-by-line debugging. This mode can be used to debug the Open AT Application by inserting various breakpoints and traces. Dynamic changes in the contents of the memory can be viewed using Developer Studio.

Sierra Wireless provides the RTE (Remote Task Execution) tool to debug the Open AT Application in remote mode.
The following chapters of this document provide more information on the target and remote modes of Open AT Application development.

Summary

The following points have been covered in this chapter:

- **The embedded Software Architecture includes the Firmware and the Open AT Application.**
- **OASiS is the set of embedded software components provided by Sierra Wireless to build an Open AT application**
- **The Download over the Air (DOTA I/II/III) service is provided by Open AT OS to upgrade the existing/running Open AT Applications and Firmware dynamically over the air (using a GSM/GPRS link).**
- **Open AT OS provides ADL Interface for Open AT Application development.**
- **Based on the memory embedded module type of Sierra product, different sizes of Open AT Application can be downloaded in the embedded module.**

CHAPTER 4

Software Installation

1. Objective

This chapter guides you through the software installation needed to develop your Open AT application.

2. What is Sierra Wireless Software Suite?

Sierra Wireless Software Suite is an application development framework that allows designing an Open AT application.

Sierra Wireless Software Suite includes:

- An IDE (Integrated Development Environment) named Developer Studio, which has been developed on top of Eclipse-provided tools (C/C++ Development Tools, Remote System Explorer) to support development steps for designing an Open AT Application
- The set of embedded module software components packaged into the Open AT Embedded Software Suite (OASiS)
- A optional set of drivers to allow the communication between the PC and the embedded module

3. System Requirements

To be able to successfully install Developer Studio, your system must support one of the following Operating Systems:

- WINDOWS XP
- WINDOWS VISTA
- WINDOWS 7

In countries with double-byte operating systems (CJK), Developer studio installation may require installation of Java package. Please contact your Sierra Wireless distributor for more information.

NOTE:
CJK – Chinese Japanese Korean

4. Hardware Requirements

The main requirements for the PC to support the Developer Studio are:

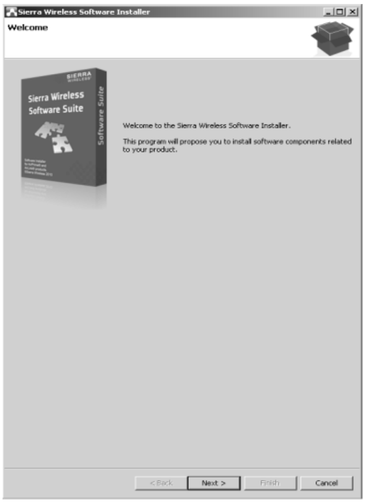
- Pentium 1GHz (or higher) processor with at least 1 GB of RAM and 500 MB free hard disk space (Recommended: dual core 2GHz processor or higher, RAM: 2GB or more).
- At least, one free COM (serial) port for communication with the target product.

5. Installing Sierra Wireless Software Suite

Please, download the latest version of Sierra Wireless Software Suite from the Sierra Wireless Web Site at http://www.sierrawireless.com. After the 1st installation, the new versions of Developer Studio and OASiS will be automatically notified to you for update.

The steps listed below will guide you through the installation of all Sierra Wireless Software Suite including Developer Studio, OASiS and drivers. Refer to the screenshots for references made to specific field labels.

1. Launch the executable file from the location of its storage (CD ROM or USB Flash drive). You will get this screen. Click **Next** to continue.

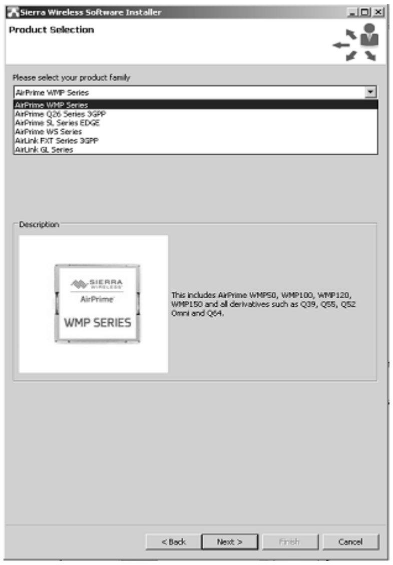

Sierra Wireless Software Installer
Welcome
Sierra Wireless Software Suite
Software Suite
< Back
Next >
Finish
Cancel

Welcome to the Sierra Wireless Software Installer.

This program will propose you to install software components related to your product.

Figure 6 – Sierra Wireless Software Suite Installation Wizard

2. The Product Selection screen is displayed. Select your Product Family and click Next to continue.


Sierra Wireless Software Installer

Product Selection

Please select your product family

AirPrime WMP Series

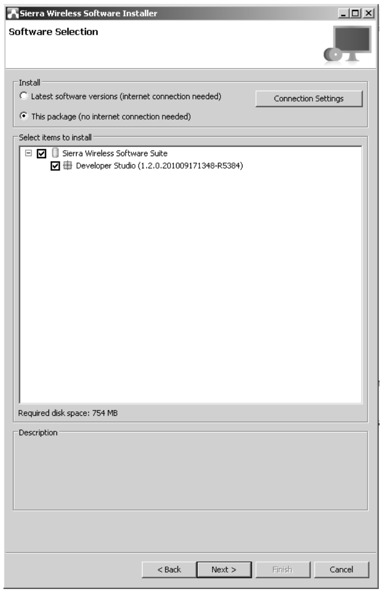
- AirPrime WMP Series
- AirPrime Q26 Series 3GPP
- AirPrime SL Series EDGE
- AirPrime WS Series
- AirLink FXT Series 3GPP
- AirLink GL Series

Description

This includes AirPrime WMP50, WMP100, WMP120, WMP150 and all derivatives such as Q39, Q55, Q52 Omni and Q64.

Figure 7 – Sierra Wireless Software Suite Installation Wizard

3. Select the items to install in the Software Selection screen. Click Next to continue.


Sierra Wireless Software Installer

Software Selection

Install

( ) Latest software versions (internet connection needed)

(•) This package (no internet connection needed)

Select items to install

- [✓] Sierra Wireless Software Suite
  - [✓] Developer Studio (1.2.0.201009171348-R5384)

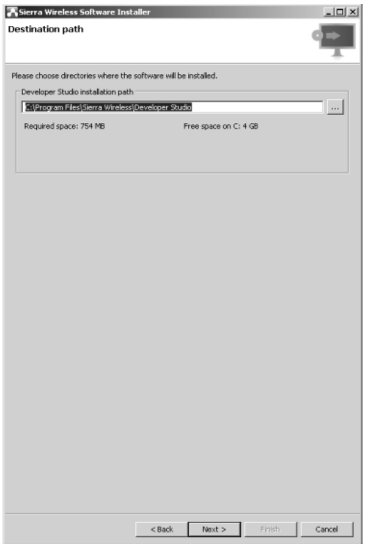
Required disk space: 754 MB

Description

Figure 8 – Sierra Wireless Software Suite Installation Wizard

4. Select the destination directory where the Sierra Wireless Software Suite components will be installed.

   - Click on Browse to select the destination folder from the folders tree. It can also be entered manually. If the destination folder does not exist, it will be created automatically before extraction.


Sierra Wireless Software Installer

Destination path

Please choose directories where the software will be installed.

Developer Studio installation path

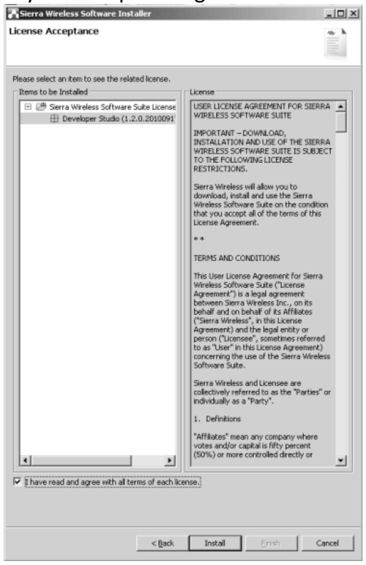
C:\Program Files\Sierra Wireless\Developer Studio

Required space: 754 MB   Free space on C: 4 GB

Figure 9 – Sierra Wireless Software Suite Installation Wizard

- Click **Next** to continue.

5. The License Agreement screen is displayed. Accept the agreement and click Install to continue.


Sierra Wireless Software Installer

License Acceptance

Please select an item to see the related license.

Items to be Installed

Sierra Wireless Software Suite License

Developer Studio (1.2.0.2010091

License

USER LICENSE AGREEMENT FOR SIERRA WIRELESS SOFTWARE SUITE

IMPORTANT – DOWNLOAD, INSTALLATION AND USE OF THE SIERRA WIRELESS SOFTWARE SUITE IS SUBJECT TO THE FOLLOWING LICENSE RESTRICTIONS.

Sierra Wireless will allow you to download, install and use the Sierra Wireless Software Suite on the condition that you accept all of the terms of this License Agreement.

* *

TERMS AND CONDITIONS

This User License Agreement for Sierra Wireless Software Suite ("License Agreement") is a legal agreement between Sierra Wireless Inc., on its behalf and on behalf of its Affiliates ("Sierra Wireless", in this License Agreement) and the legal entity or person ("Licensee", sometimes referred to as "User" in this License Agreement) concerning the use of the Sierra Wireless Software Suite.

Sierra Wireless and Licensee are collectively referred to as the "Parties" or individually as a "Party".

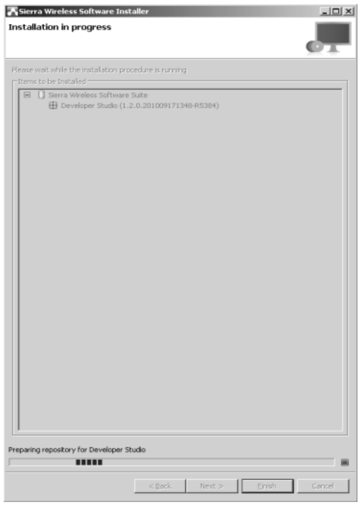
1. Definitions

"Affiliates" mean any company where votes and/or capital is fifty percent (50%) or more controlled directly or

[✓] I have read and agree with all terms of each license.

Figure 10 – Sierra Wireless Software Suite Installation Wizard

6. The extraction is automatically started once you click Install and the following screen appears.


Sierra Wireless Software Installer

Installation in progress

Please wait while the installation procedure is running

Items to be Installed

Sierra Wireless Software Suite
Developer Studio (1.2.0.201009171348-R5384)

Preparing repository for Developer Studio

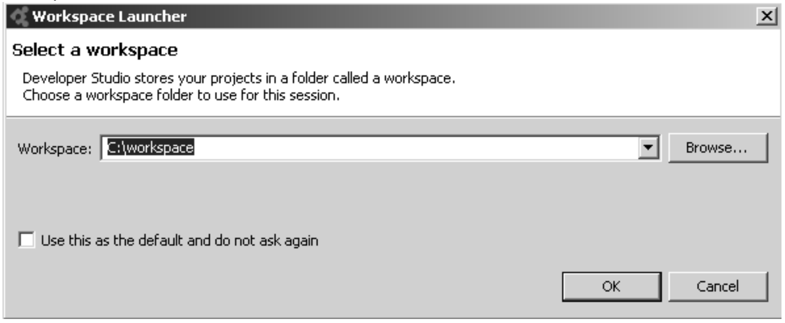
Figure 11 - Sierra Wireless Software Suite Installation Wizard

7. Once the package is successfully extracted, the setup propose you to launch Developer Studio automatically and you will be asked to choose a workspace folder.

Figure 12 – Developer Studio

- Click on Browse to select the workspace folder where Open AT packages/projects should be stored. The workspace selected here is applicable for the current session. In case you want to use this as the default workspace for importing Software Suite packages and for creating Open AT Applications, then mark the checkbox “Use this as default and do not ask again”.

- After the workspace is selected, click on OK.


Workspace Launcher

Select a workspace

Developer Studio stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

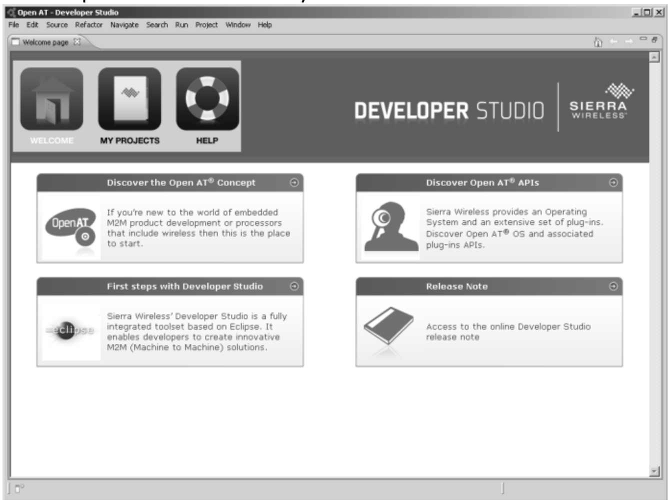
Workspace: C:\workspace

☐ Use this as the default and do not ask again

Figure 13 – Select Workspace for Developer Studio

NOTE:
The workspace path shall not contain space character.
The workspace path length shall be less than 50 characters.

8. Once the workspace path is specified it automatically re-directs you to the Welcome Page of Developer studio which means that the Developer studio is now ready to use.


Open AT - Developer Studio
File Edit Source Refactor Navigate Search Run Project Window Help
Welcome page
WELCOME
MY PROJECTS
HELP
DEVELOPER STUDIO
SIERRA WIRELESS

Discover the Open AT® Concept
If you're new to the world of embedded M2M product development or processors that include wireless then this is the place to start.

Discover Open AT® APIs
Sierra Wireless provides an Operating System and an extensive set of plug-ins. Discover Open AT® OS and associated plug-ins APIs.

First steps with Developer Studio
Sierra Wireless' Developer Studio is a fully integrated toolset based on Eclipse. It enables developers to create innovative M2M (Machine to Machine) solutions.

Release Note
Access to the online Developer Studio release note

Figure 14 – Developer Studio Welcome Page

6. Developer Studio Update

Once installed, Developer Studio will automatically notify you when a new Developer Studio version is available and will propose you to install it.

7. Software Package Manager

Developer Studio owns a Software Package Manager perspective which allows users to install or update any embedded software components provided by Sierra Wireless such as the OASiS. This perspective allows also installing or updating any embedded software components provided by a third party like a plug-in for Open AT.

To be able to use OASiS with Developer Studio, you will have to follow the below mentioned steps:

- You will have to first install the OASiS package in Developer Studio either through the Sierra Wireless Software Suite installation process or the Developer Studio Software Package Manager Installation process. This will extract the contents of the packages in the Developer Studio workspace, making them available to the developer for the Open AT Application.
- The OASiS needs to be imported only once in Developer Studio. Next time when an Open AT Application is to be created, you can use the previously imported package for the same. If a different version of OASiS is required, then the same has to be imported in the Developer Studio workspace before using it for Open AT Application development.

- If more than one OASiS has been installed, you can select the version to be used from the available list when creating an Open AT Application.
- After importing an OASiS, the Open AT perspective available in Developer Studio can be used to create/execute/debug Open AT Applications.

The detailed process to be followed to do the above steps is explained in the following sections of this chapter.

7.1. Importing OASiS in Developer Studio

Before you begin developing an Open AT Application, you will have to import the OASiS package in the Developer Studio workspace. If it has not been previously installed through the Sierra Wireless Software Suite installation process, the following steps will guide you through the process of importing the OASiS in Developer Studio.

1. Select Window > Open Perspective > Other > Packages Manager and then click **OK.**

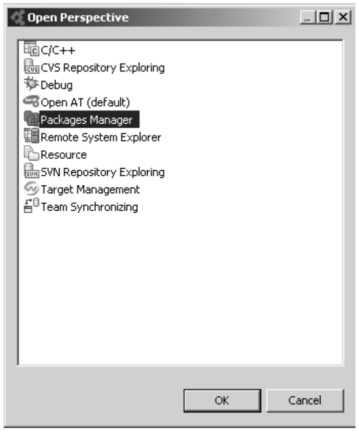
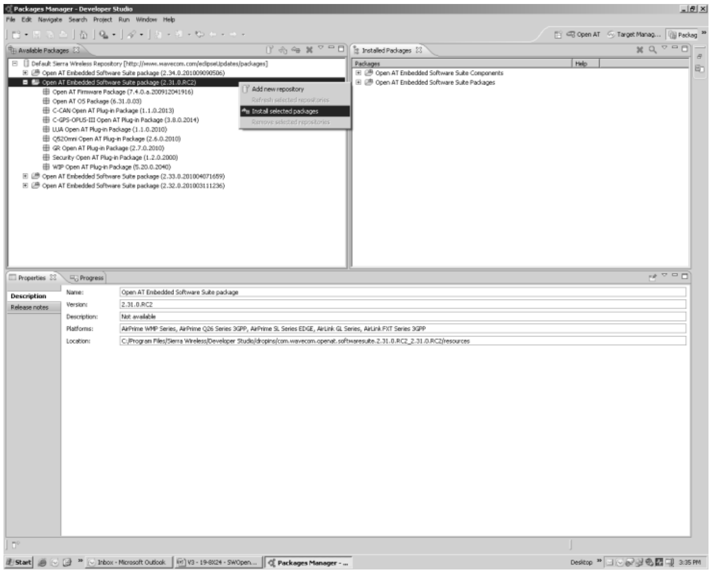


Figure 15 – Open AT perspective

2. In the Packages Manager perspective look for Available Packages window. Select the package you want to install, right click and select “Install selected packages“.


Packages Manager - Developer Studio
File Edit Navigate Search Project Run Window Help
Available Packages
Installed Packages
Add new repository
Refresh selected repositories
Install selected packages
Remove selected repositories
Properties
Progress
Description
Release notes

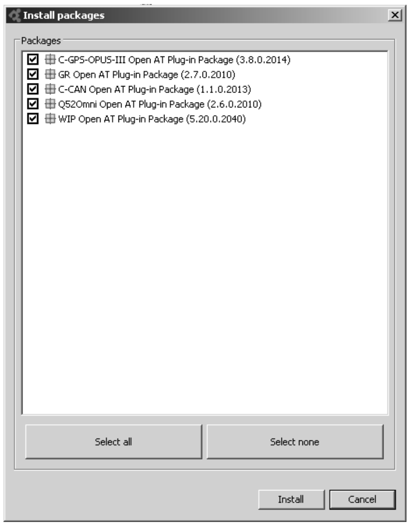
Default Sierra Wireless Repository [http://www.wavecom.com/eclipseUpdates/packages]
- Open AT Embedded Software Suite package (2.34.0.201009090506)
- Open AT Embedded Software Suite package (2.31.0.RC2)
  - Open AT Firmware Package (7.4.0.a.200912041916)
  - Open AT OS Package (6.31.0.03)
  - C-CAN Open AT Plug-in Package (1.1.0.2013)
  - C-GPS-OPUS-III Open AT Plug-in Package (3.8.0.2014)
  - LUA Open AT Plug-in Package (1.1.0.2010)
  - Q52Omni Open AT Plug-in Package (2.6.0.2010)
  - GR Open AT Plug-in Package (2.7.0.2010)
  - Security Open AT Plug-in Package (1.2.0.2000)
  - WIP Open AT Plug-in Package (5.20.0.2040)
- Open AT Embedded Software Suite package (2.33.0.201004071659)
- Open AT Embedded Software Suite package (2.32.0.201003111236)

Packages
- Open AT Embedded Software Suite Components
- Open AT Embedded Software Suite Packages

Name: Open AT Embedded Software Suite package
Version: 2.31.0.RC2
Description: Not available
Platforms: AirPrime WMP Series, AirPrime Q26 Series 3GPP, AirPrime SL Series EDGE, AirLink GL Series, AirLink FXT Series 3GPP
Location: C:/Program Files/Sierra Wireless/Developer Studio/dropins/com.wavecom.openat.softwaresuite.2.31.0.RC2_2.31.0.RC2/resources

Figure 16 – Packages Manager Perspective

3. The Install Packages window is displayed. Click on **Select all** and then **Install**.



- [✓] C-GPS-OPUS-III Open AT Plug-in Package (3.8.0.2014)
- [✓] GR Open AT Plug-in Package (2.7.0.2010)
- [✓] C-CAN Open AT Plug-in Package (1.1.0.2013)
- [✓] Q52Omni Open AT Plug-in Package (2.6.0.2010)
- [✓] WIP Open AT Plug-in Package (5.20.0.2040)

Figure 17 – Install Packages Window

4. Once the installation is complete, you will be directed to the ‘Open AT Perspective’ in Developer Studio and an Open AT Application can now be created using Developer Studio.

Summary

The following points have been covered in this chapter:

- **Sierra Wireless Software Suite includes an IDE named Developer Studio, the embedded Software components named OASiS and drivers**
- **The installation of Sierra Wireless Software Suite**
- **The Software Package Manager allows to import OASiS or any third party plug-in into the Developer Studio**

CHAPTER 5

Open AT Project Development

1. Objective

This chapter describes how to develop an Open AT project using Developer Studio.
Developer Studio is the IDE allowing you to develop your Open AT application. It supports you during these following steps:

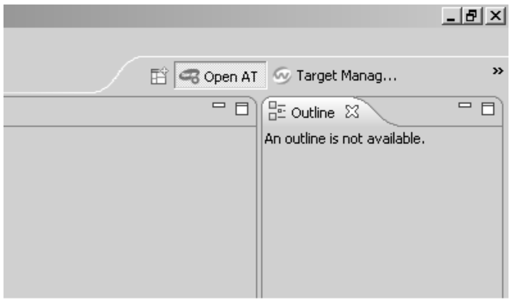
- Create an Open AT project
- Develop an Open AT application
- Compile and link the Open AT application
- Download an Open AT application into the embedded module
- Execute an Open AT application in Target or Remote mode.
- Debug an Open AT application.

2. Creating an Open AT Application Project using Developer Studio

You can create an Open AT Application project using the Developer Studio.

2.1. Using Developer Studio to create Open AT Application

1. Select the 'Open AT Perspective' from the top right tabs in Developer Studio.

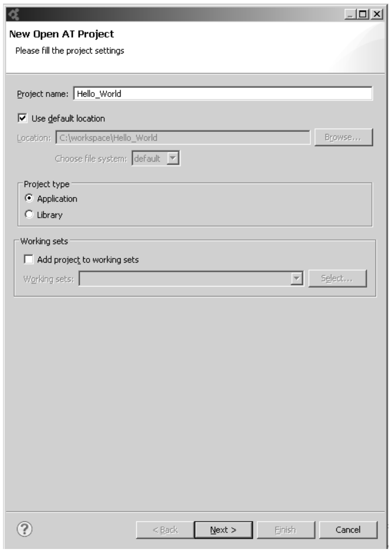


An outline is not available.

Figure 18 – Open AT perspective

2. Select File > New > Open AT Project from the Developer Studio menu. This will take you to the Open AT project creation wizard.
3. By default, the new Open AT Application will be created in the workspace provided during the installation of Developer Studio. If you wish to change the Open AT Application location, uncheck the box which says "Use default location" and specify the location manually.

4. Provide the project name and select the project type. You can select either of the project type.


New Open AT Project
Please fill the project settings

Project name: Hello_World

☑ Use default location

Location: C:\workspace\Hello_World

Choose file system: default

Project type
- (●) Application
- ( ) Library

Working sets
- ☐ Add project to working sets

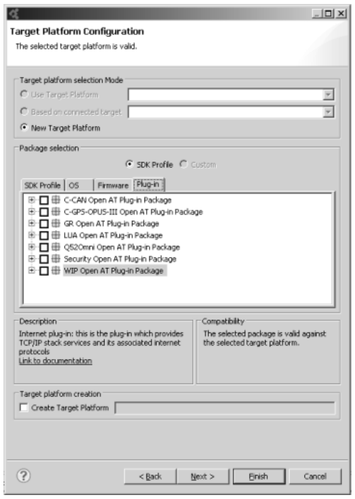
Working sets:

Figure 19 – Open AT perspective

5. Click **Next** to continue.

6. The Target Platform Configuration window will be displayed. In the package selection:

- Select the appropriate Open AT Plug-in to be used in the Open AT Application under Plug-in tab. You can select one or more depending on the project requirements.
- Select the Open AT OS package that should be used to build the Open AT Application under OS tab.
- Select the Open AT Firmware package that should be used under Firmware tab.
- You can either press **Finish** to create a new empty template project, or **Next** to go to the sample selection step.
- The next step allows configuring the sample on which the project will be based.


Target Platform Configuration
The selected target platform is valid.

Target platform selection Mode
( ) Use Target Platform
( ) Based on connected target
(•) New Target Platform

Package selection
(•) SDK Profile ( ) Custom

SDK Profile | OS | Firmware | Plug-in

- ☐ C-CAN Open AT Plug-in Package
- ☐ C-GPS-OPUS-III Open AT Plug-in Package
- ☐ GR Open AT Plug-in Package
- ☐ LUA Open AT Plug-in Package
- ☐ Q520mni Open AT Plug-in Package
- ☐ Security Open AT Plug-in Package
- ☐ WIP Open AT Plug-in Package

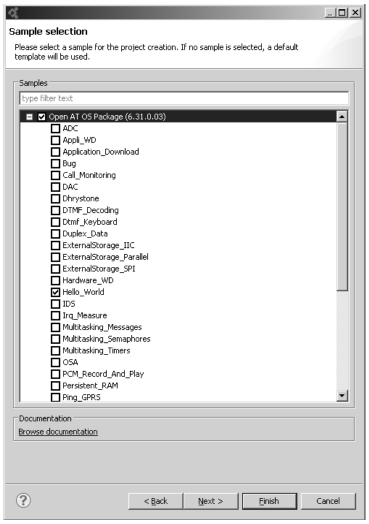
Description
Internet plug-in: this is the plug-in which provides TCP/IP stack services and its associated internet protocols
Link to documentation

Compatibility
The selected package is valid against the selected target platform.

Target platform creation
☐ Create Target Platform

Figure 20 - Open AT project creation wizard

7. Sample selection window will be displayed, select a sample for the project creation. If no sample is selected, a default template will be used.


Sample selection
Please select a sample for the project creation. If no sample is selected, a default template will be used.

Samples
type filter text

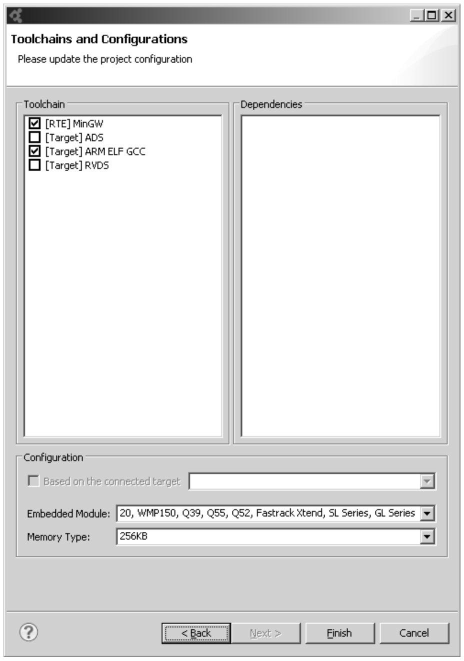
- ☑ Open AT OS Package (6.31.0.03)
  - ☐ ADC
  - ☐ Appli_WD
  - ☐ Application_Download
  - ☐ Bug
  - ☐ Call_Monitoring
  - ☐ DAC
  - ☐ Dhrystone
  - ☐ DTMF_Decoding
  - ☐ Dtmf_Keyboard
  - ☐ Duplex_Data
  - ☐ ExternalStorage_IIC
  - ☐ ExternalStorage_Parallel
  - ☐ ExternalStorage_SPI
  - ☐ Hardware_WD
  - ☑ Hello_World
  - ☐ IDS
  - ☐ Irq_Measure
  - ☐ Multitasking_Messages
  - ☐ Multitasking_Semaphores
  - ☐ Multitasking_Timers
  - ☐ OSA
  - ☐ PCM_Record_And_Play
  - ☐ Persistent_RAM
  - ☐ Ping_GPRS

Documentation
Browse documentation

Figure 21 – Open AT perspective

8. Select the appropriate toolchain to be used. You can select one or more depending on the project requirements. The Dependencies list allows adding references to already existing library projects in the workspace. The Configuration group allows to set some Target related additional parameters:

   - The embedded Module, when using a Target Platform supporting several types
   - The Memory Type (for Open AT Application projects only), to configure the memory settings according to the used target type
   - Click **Finish** to continue.


Toolchains and Configurations

Please update the project configuration

Toolchain
- [x] [RTE] MinGW
- [ ] [Target] ADS
- [x] [Target] ARM ELF GCC
- [ ] [Target] RVDS

Dependencies

Configuration

- [ ] Based on the connected target

Embedded Module: 20, WMP150, Q39, Q55, Q52, Fastrack Xtend, SL Series, GL Series

Memory Type: 256KB

Figure 22 – Toolchains and Configurations window

NOTE:
A new compiler will be available in Toolchain for SDK Profile 2.34 and above, ARM EABI GCC. When an Open AT project is created this compiler is selected by default, user can also select all other compilers as the previous compiler list will still be available.

9. The Open AT Project is created and the screen is directed to the ‘Open AT perspective’. The Open AT Project is displayed in the Project Explorer pane.

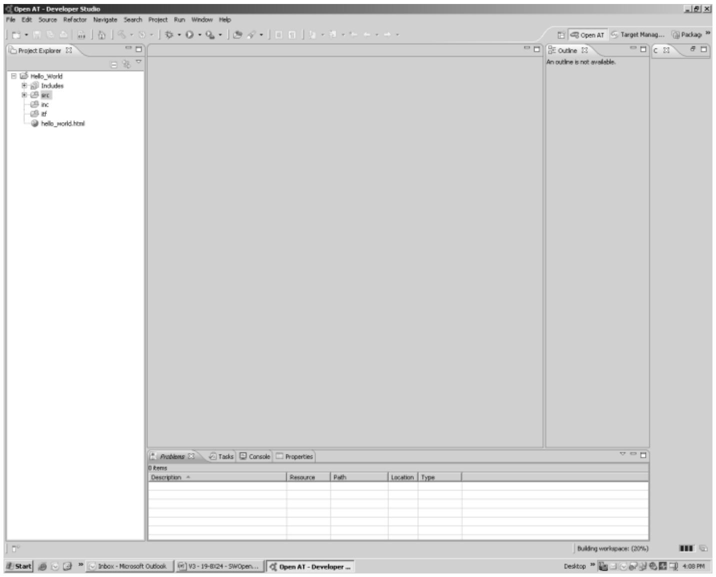


Figure 23 – Open AT perspective

2.2. Compiling and executing the project

Open AT project can be compiled and executed in either of the following modes:

- Target mode
- Remote mode

2.2.1. Target Mode

To run an Open AT Application in the target or download mode, you must build the Open AT Application in Target mode and create a Run Configuration accordingly.

To compile and build the Open AT Application Developer Studio, follow the steps below:

1. To build an Open AT Application you will have to select the mode in which the Open AT Application should be built. This can be done through Project > Build Configurations > Set Active and select either Target Release or Target Debug configuration.

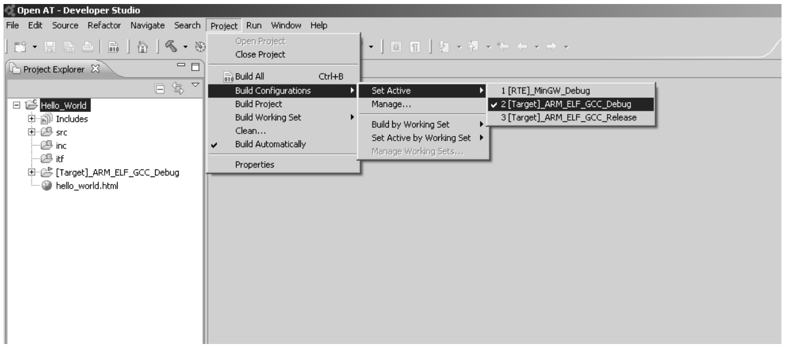


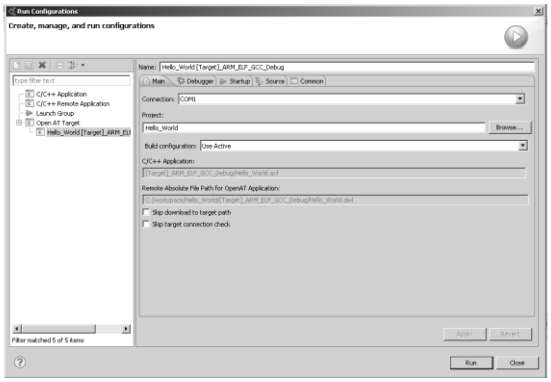
Figure 24 – Select Target Build Configuration

2. The Target Release mode should be selected if no debug information is required from the Open AT Application. When this mode is selected, the Traces used in the Open AT Application are not logged in the Trace View window of Developer Studio.

3. The Target Debug mode should be selected when the debug information provided in the Open AT Application is required. When this mode is selected, the Traces used in the Open AT Application are logged in the Trace View window of Developer Studio.

4. After selecting the build mode, select Project > Build Project to build the project.

5. Create a Run Configuration to execute the Open AT Application. To do this, select the project root folder in the Project Explorer pane and right click on it. Then select Run As > Run Configurations. This will open a Run Configurations window.



Name: Hello_World [Target]_ARM_ELF_GCC_Debug

Connection: COM1

Project: Hello_World

Build configuration: Use Active

C/C++ Application: [Target]_ARM_ELF_GCC_Debug/Hello_World.axf

Remote Absolute File Path for OpenAT Application: C:/workspace/Hello_World/[Target]_ARM_ELF_GCC_Debug/Hello_World.dwl

☐ Skip download to target path

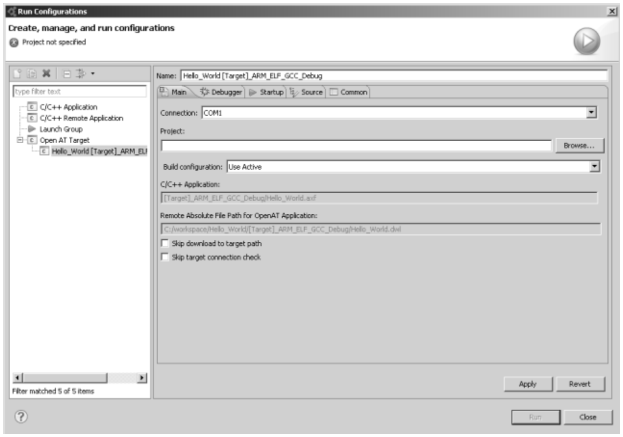
☐ Skip target connection check

Figure 25 – Open AT Target Run Configuration

To create a new Open AT Target configuration, double click on Open AT Target option on the left pane.

Notes:
A Run Configuration can be re-used for the same Open AT project. If a Run Configuration already exists for an Open AT Application, there is no need to create a new Run Configuration every time you want to execute the Open AT Application.

6. Once the Run Configuration is created successfully, all the fields are filled automatically by Developer Studio. In case of an error, an error message is displayed on the top left of the Run Configurations window.


Run Configurations
Create, manage, and run configurations
Project not specified

Name: Hello_World [Target]_ARM_ELF_GCC_Debug
Connection: COM1
Project:
Build configuration: Use Active
C/C++ Application: [Target]_ARM_ELF_GCC_Debug/Hello_World.axf
Remote Absolute File Path for OpenAT Application: C:/workspace/Hello_World/[Target]_ARM_ELF_GCC_Debug/Hello_World.dwl
☐ Skip download to target path
☐ Skip target connection check

Figure 26 – Open AT Target Run Configuration error

7. If the Run Configuration is successful, click on Run to execute the Open AT Application. In case of an error, delete the incorrect configuration and check for the specified error before creating a new one.

8. After executing a successful Run Configuration, you can switch to the 'Target Management' perspective and communicate with the embedded module.

2.2.2. Remote Mode

In the remote mode, the Open AT Application can be compiled and executed in M2M Studio using the following steps:

1. Set active the mode that should be used to build the Open AT Application. This can be done through Project > Build Configurations > Set Active and selecting RTE Debug configuration. RTE debug configuration should be selected if you want to debug the Open AT Application by putting break points.

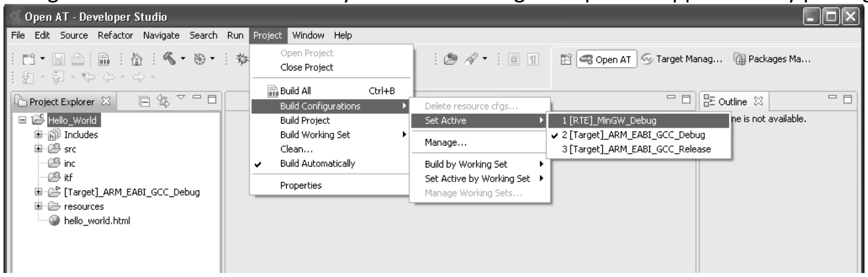
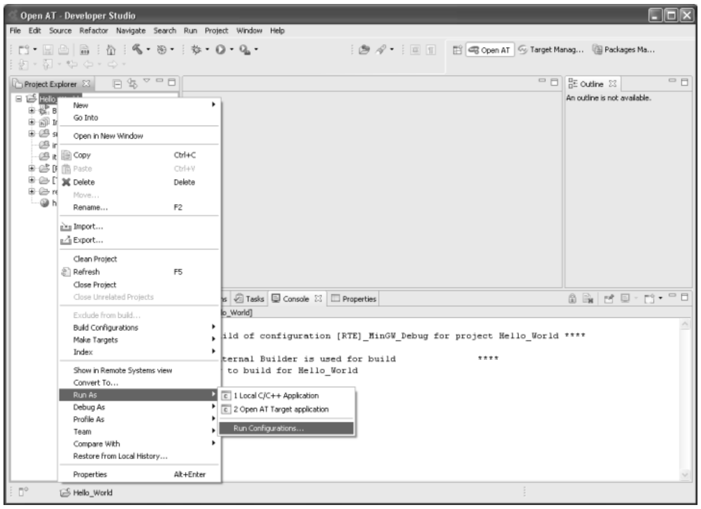


Figure 27 – Select Remote Build Configuration

2. Select Project > Build Project to build the project.

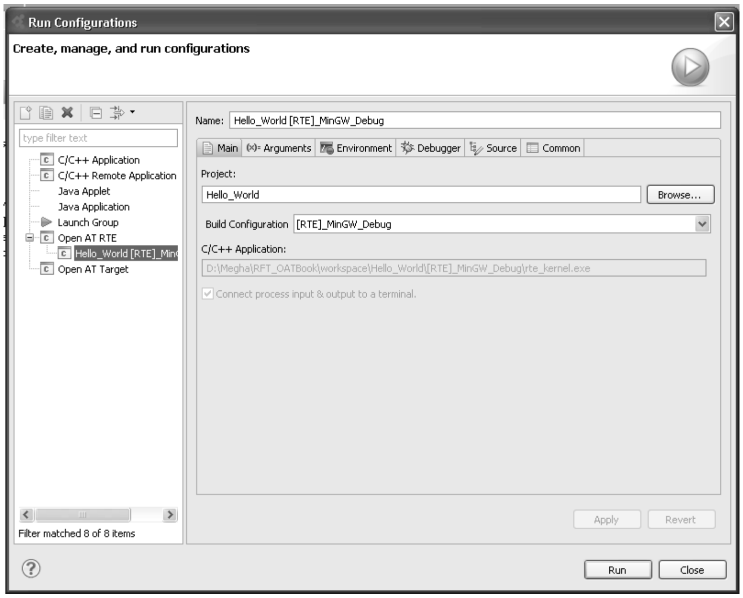
3. Create a Run Configuration to execute the Open AT Application in RTE mode. To do this, select the project root folder in the Project Explorer pane and right click on it. Then select Run As > Run

Configurations. This will open a Run Configurations window. Create a new Open AT RTE configuration by double clicking on Open AT RTE option on the left pane.



```
ild of configuration [RTE]_MinGW_Debug for project Hello_World ****

ternal Builder is used for build             ****
 to build for Hello_World
```

Figure 28 – Run Configuration



Name: Hello_World [RTE]_MinGW_Debug

Project: Hello_World

Build Configuration [RTE]_MinGW_Debug

C/C++ Application: D:\Megha\RFT_OATBook\workspace\Hello_World\[RTE]_MinGW_Debug\rte_kernel.exe

✓ Connect process input & output to a terminal.

Filter matched 8 of 8 items

Figure 29 – Open AT RTE Run Configuration Window

4. Once the Run Configuration is created successfully, all the fields are filled automatically by M2M Studio. Click on Run to start executing the Open AT Application in RTE mode. This will open the Remote Task Monitor window.

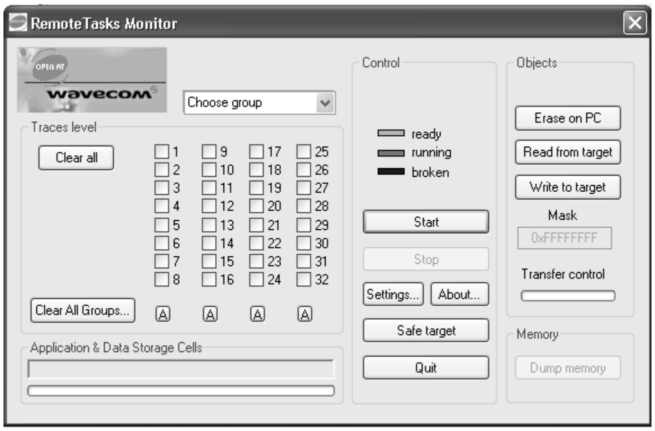


Figure 30 – Remote Tasks Monitor

5. Select the ADL traces from the drop down menu and check the required trace levels. Click on Start to start executing the Open AT Application in Remote mode.

6. Once the Open AT Application execution is complete, click Stop to stop the Remote Task Monitor. This option is enabled only if the Start option is disabled.

7. Click Safe target and Quit to exit the remote mode of the Open AT Application.

2.3. Debugging Open AT Application in Remote mode

To debug an Open AT Application, it should be executed in Remote mode by selecting the appropriate debug configuration. The following steps will guide you through debugging an Open AT Application in Remote mode using M2M Studio.

1. Set active RTE mode to be used to build the Open AT Application. This can be done through Project > Build Configurations > Set Active and select RTE Debug configuration.

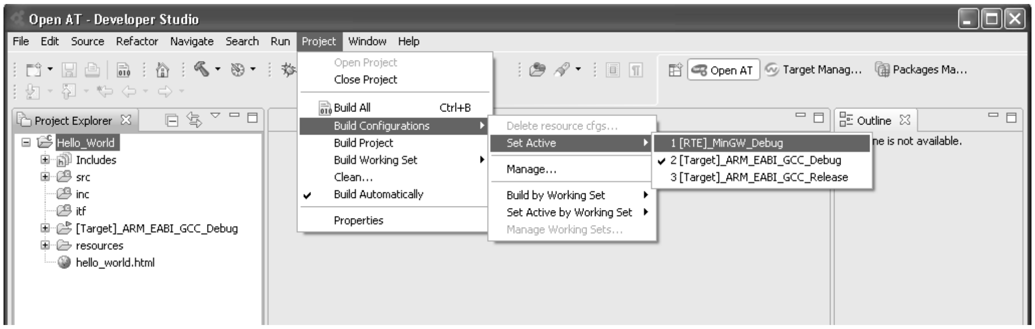
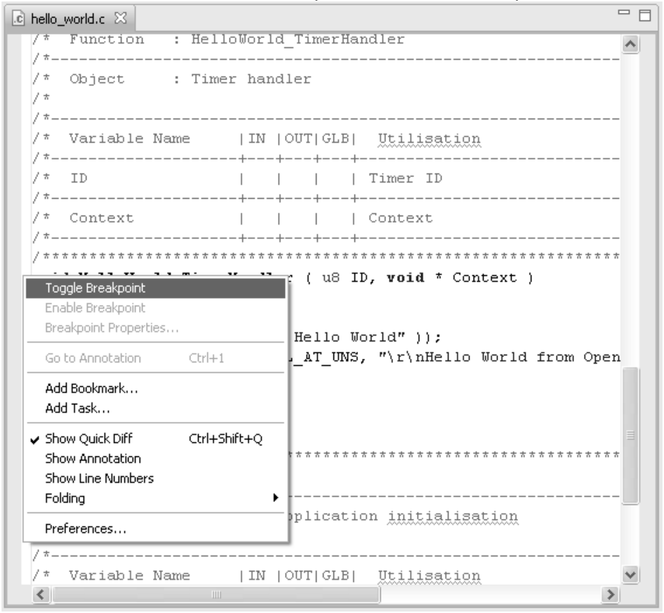


Figure 31 – Select Remote Debug Build Configuration

2. Select Project > Build Project to build the project.

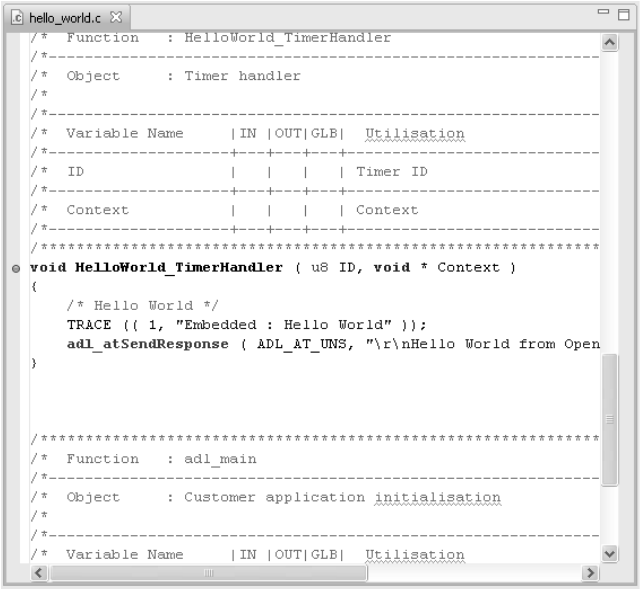
3. To set breakpoint, right-click in the marker bar area on the left side of the editor beside the line where the program to be suspended, then choose Toggle Breakpoint option. You can also double click on the marker bar next to the source code line to put/remove a break point.



```
/*  Function   : HelloWorld_TimerHandler
/*--------------------------------------------------------------
/*  Object     : Timer handler
/*
/*--------------------------------------------------------------
/*  Variable Name     |IN |OUT|GLB|  Utilisation
/*--------------------+---+---+---+-----------------------------
/*  ID                |   |   |   | Timer ID
/*--------------------+---+---+---+-----------------------------
/*  Context           |   |   |   | Context
/*--------------------+---+---+---+-----------------------------
/***************************************************************
                      ( u8 ID, void * Context )

                      Hello World" ));
                      _AT_UNS, "\r\nHello World from Open


                      **************************************

                      --------------------------------------
                      plication initialisation
/*--------------------------------------------------------------
/*  Variable Name     |IN |OUT|GLB|  Utilisation
```

Figure 32 – Insert break point

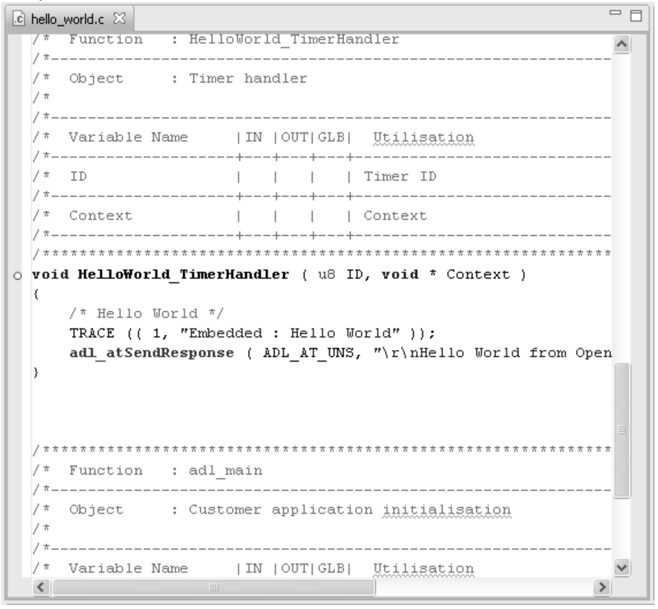
4. Enabled breakpoints are indicated with a blue circle as shown below.



```
/*  Function   : HelloWorld_TimerHandler
/*--------------------------------------------------------------
/*  Object     : Timer handler
/*
/*--------------------------------------------------------------
/*  Variable Name     |IN |OUT|GLB|  Utilisation
/*--------------------+---+---+---+-----------------------------
/*  ID                |   |   |   | Timer ID
/*--------------------+---+---+---+-----------------------------
/*  Context           |   |   |   | Context
/*--------------------+---+---+---+-----------------------------
/***************************************************************
void HelloWorld_TimerHandler ( u8 ID, void * Context )
{
    /* Hello World */
    TRACE (( 1, "Embedded : Hello World" ));
    adl_atSendResponse ( ADL_AT_UNS, "\r\nHello World from Open
}



/***************************************************************
/*  Function   : adl_main
/*--------------------------------------------------------------
/*  Object     : Customer application initialisation
/*
/*--------------------------------------------------------------
/*  Variable Name     |IN |OUT|GLB|  Utilisation
```

Figure 33 – Enabled break point

5. Disabled breakpoints are indicated with a white circle as shown below.

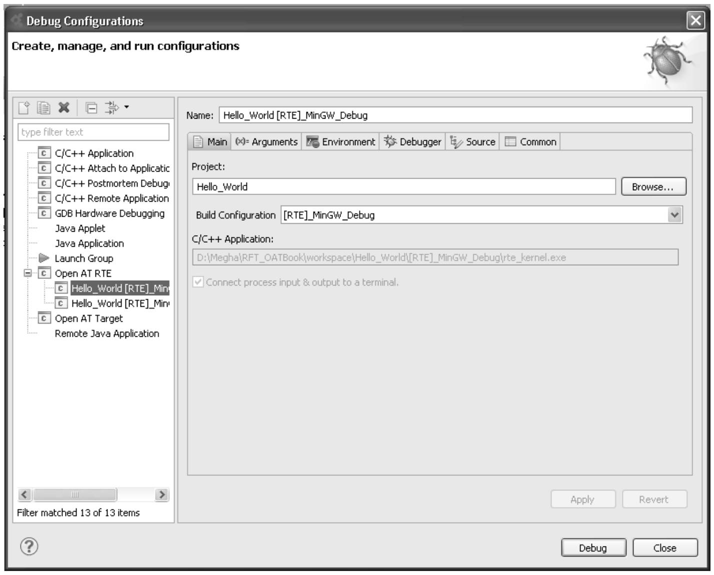


hello_world.c

```
/*  Function   : HelloWorld_TimerHandler
/*---------------------------------------------------------------
/*  Object     : Timer handler
/*
/*---------------------------------------------------------------
/*  Variable Name     |IN |OUT|GLB|  Utilisation
/*--------------------+---+---+---+-----------------------------
/*  ID                |   |   |   | Timer ID
/*--------------------+---+---+---+-----------------------------
/*  Context           |   |   |   | Context
/*--------------------+---+---+---+-----------------------------
/****************************************************************
void HelloWorld_TimerHandler ( u8 ID, void * Context )
{
    /* Hello World */
    TRACE (( 1, "Embedded : Hello World" ));
    adl_atSendResponse ( ADL_AT_UNS, "\r\nHello World from Open
}



/****************************************************************
/*  Function   : adl_main
/*---------------------------------------------------------------
/*  Object     : Customer application initialisation
/*
/*---------------------------------------------------------------
/*  Variable Name     |IN |OUT|GLB|  Utilisation
```

Figure 34 – Disabled break point

6. Create a Debug Configuration to debug the Open AT Application in RTE mode. To do this, select the project root folder in the Project Explorer pane and right click on it. Then select Debug As > Debug Configurations. This will open a Debug Configurations window. Create a new Open AT RTE configuration by double clicking on Open AT RTE option on the left pane.



Debug Configurations

Create, manage, and run configurations

type filter text

- C/C++ Application
- C/C++ Attach to Applicatic
- C/C++ Postmortem Debug
- C/C++ Remote Application
- GDB Hardware Debugging
- Java Applet
- Java Application
- Launch Group
- Open AT RTE
  - Hello_World [RTE]_Min
  - Hello_World [RTE]_Min
- Open AT Target
- Remote Java Application

Filter matched 13 of 13 items

Name: Hello_World [RTE]_MinGW_Debug

Main | Arguments | Environment | Debugger | Source | Common

Project: Hello_World

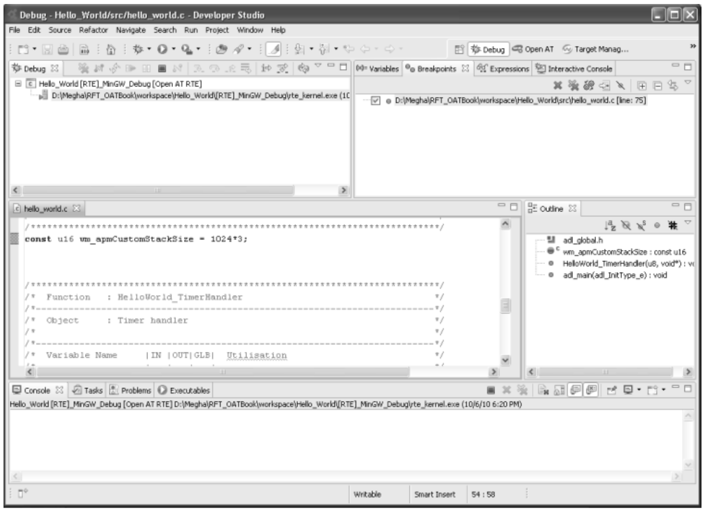
Build Configuration: [RTE]_MinGW_Debug

C/C++ Application: D:\Megha\RFT_OATBook\workspace\Hello_World\[RTE]_MinGW_Debug\rte_kernel.exe

[✓] Connect process input & output to a terminal.

Figure 35 – Open AT RTE Debug Configuration

7. Once the Debug Configuration is created successfully, all the fields are filled automatically by M2M Studio. Click on Debug to start executing the Open AT Application in RTE mode. This will open the 'Debug Perspective' in M2M Studio which can be used to control the flow of execution using break points. It will also open the Remote Tasks Monitor.



```
Hello_World [RTE]_MinGW_Debug [Open AT RTE]
    D:\Megha\RFT_OATBook\workspace\Hello_World\[RTE]_MinGW_Debug\rte_kernel.exe (10
```

```
D:\Megha\RFT_OATBook\workspace\Hello_World\src\hello_world.c [line: 75]
```

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



/***************************************************************************/
/*  Function   : HelloWorld_TimerHandler                                   */
/*-------------------------------------------------------------------------*/
/*  Object     : Timer handler                                             */
/*                                                                         */
/*-------------------------------------------------------------------------*/
/*  Variable Name     |IN |OUT|GLB|  Utilisation                           */
```

```
adl_global.h
wm_apmCustomStackSize : const u16
HelloWorld_TimerHandler(u8, void*) : v
adl_main(adl_InitType_e) : void
```

```
Hello_World [RTE]_MinGW_Debug [Open AT RTE] D:\Megha\RFT_OATBook\workspace\Hello_World\[RTE]_MinGW_Debug\rte_kernel.exe (10/6/10 6:20 PM)
```

Figure 36 – M2M Studio Debug Perspective

8. Select the ADL traces from the drop down menu and check the required trace levels. Click on Start to start executing the Open AT Application in Remote mode. Once the break point is hit, the execution of the Open AT Application will be suspended.

9. You can use the debug options available in the 'Debug perspective' to debug the Open AT Application.

10. Once the Open AT Application debugging is complete, click Stop to stop the Remote Task Monitor. This option is enabled only if the Start option is disabled.

11. Click Safe Modem and Quit to exit the remote mode of the Open AT Application.

Summary

The following points have been covered in this chapter:

- Open AT Application can be compiled and executed in Target and Remote modes.
- Run/Debug configurations are used to Execute/Debug an Open AT Application.

CHAPTER 6

Running Open AT® Application and Using Developer Studio

1. Objective

This chapter describes Developer Studio and its use for communicating with the embedded module and executing Open AT Applications.

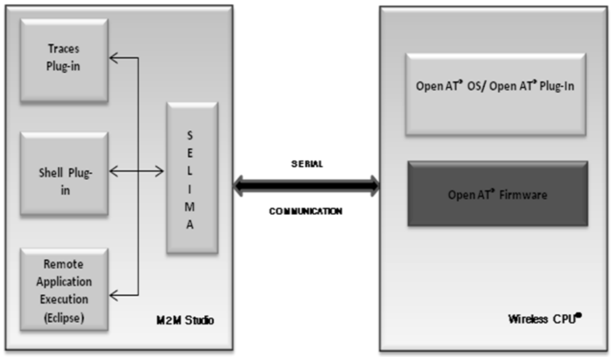


Figure 37 – Developer Studio Environment

2. Developer Studio

Developer Studio comes with built-in plug-ins that can be used during Open AT Application development and to communicate to the embedded module. Traces, Remote Shell, are some of the examples of the built-in plug-ins.

Developer Studio manages different functionalities with the help of different perspectives. The different perspectives provided by Developer Studio are:

- Target Management perspective.
- Open AT perspective.
- Debug perspective.

Each of these perspectives is covered in details in the following sections.

2.1. Target Management Perspective

The Target Management perspective is used to communicate with the embedded module over the COM port. It has built-in plug-ins to

- Displays the Open AT Application's trace messages from the target or the Remote Application.
- Send AT command and receive the corresponding response.
- Retrieve Target information.

To use Target Management perspective, follow the steps mentioned below:

1. Switch on the embedded module. Ensure the target is not in the initialization stage.

2. Make sure that the selected COM port is not in use by any other application.

3. Select the corresponding COM port under **Devices** view and use the **Change serial port** settings option to configure the COM port settings.

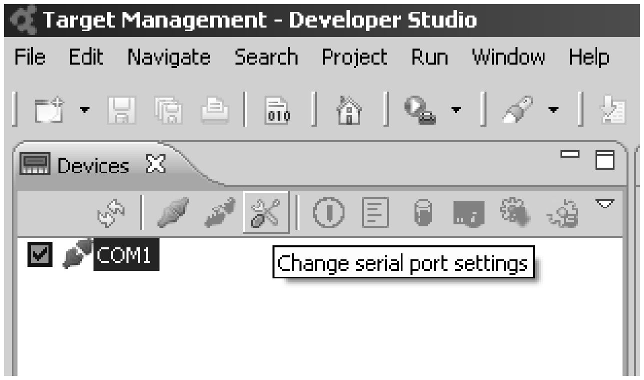
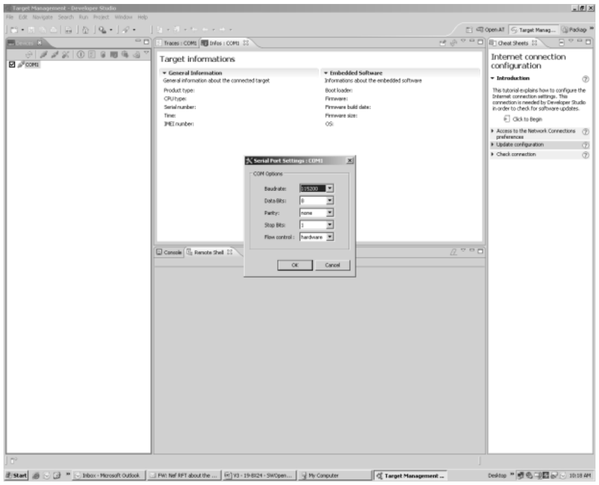

Target Management - Developer Studio
File
Edit
Navigate
Search
Project
Run
Window
Help
Devices
COM1
Change serial port settings

Figure 38 – Change serial port settings



Serial Port Settings : COM1

COM Options

Baudrate: 115200

Data Bits: 8

Parity: none

Stop Bits: 1

Flow control : hardware

Figure 39 – Serial Port settings: COM 1

4. Click on Open the selected port option under the Devices view to connect to the embedded module.

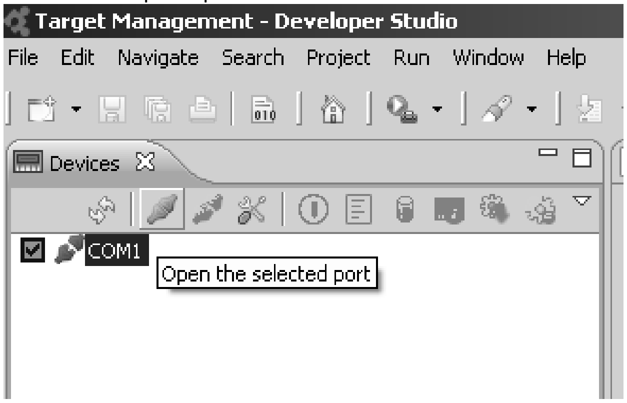

Target Management - Developer Studio
File Edit Navigate Search Project Run Window Help
Devices
COM1
Open the selected port

Figure 40 – Open the selected port

5. In case of any error while connecting, the corresponding error message will be displayed.

6. The Infos pane can be used to retrieve embedded module information. Click on the refresh button in the Infos pane to retrieve the state of the embedded module and other values.

7. Infos pane can be viewed using Window->Show View->Other. In Show View window select Target Management->Infos and then click OK.

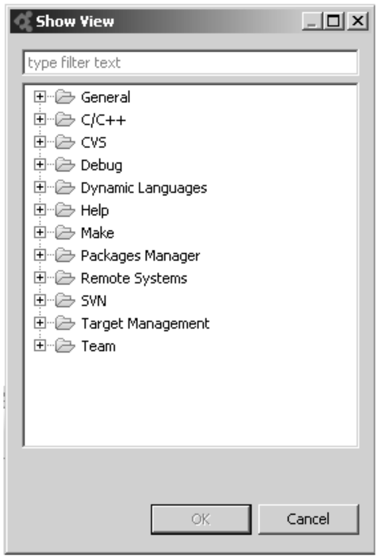
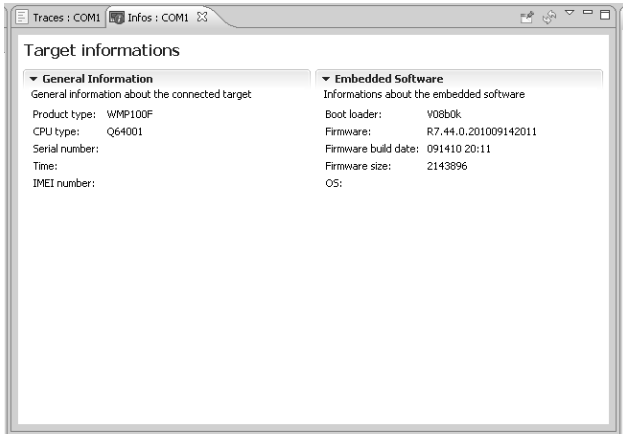


Figure 41 – Show View Window



Target informations

General Information

General information about the connected target

Product type:	WMP100F
CPU type:	Q64001
Serial number:	
Time:	
IMEI number:	

Embedded Software

Informations about the embedded software

Boot loader:	V08b0k
Firmware:	R7.44.0.201009142011
Firmware build date:	091410 20:11
Firmware size:	2143896
OS:	

Figure 42 – Infos pane (Target Information)

8. Trace View window can be viewed using Window->Show View->Other. In Show View window select Target Management->Traces and then click OK. It displays the traces logged by Open AT Application. The level of traces to be displayed can be selected by using the "Remote Traces configuration" option in the Trace View toolbar.

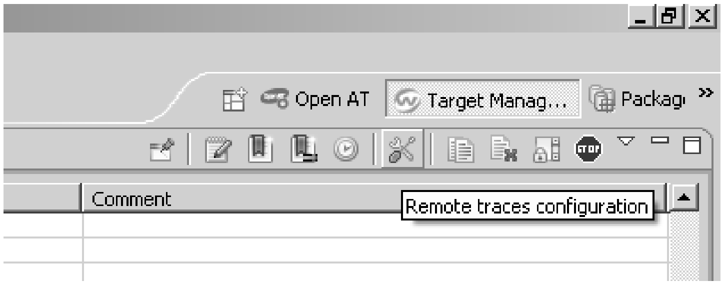
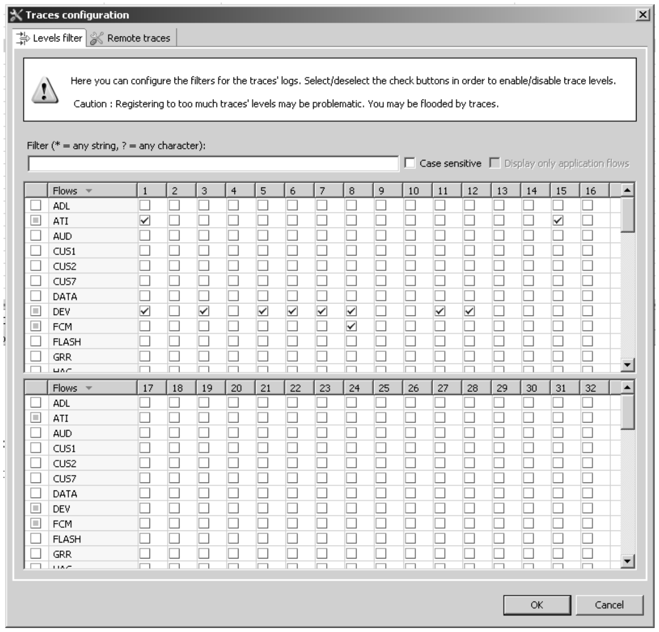


Figure 43 – Remote traces configuration



Here you can configure the filters for the traces' logs. Select/deselect the check buttons in order to enable/disable trace levels.

Caution : Registering to too much traces' levels may be problematic. You may be flooded by traces.

Filter (* = any string, ? = any character):

☐ Case sensitive ☐ Display only application flows

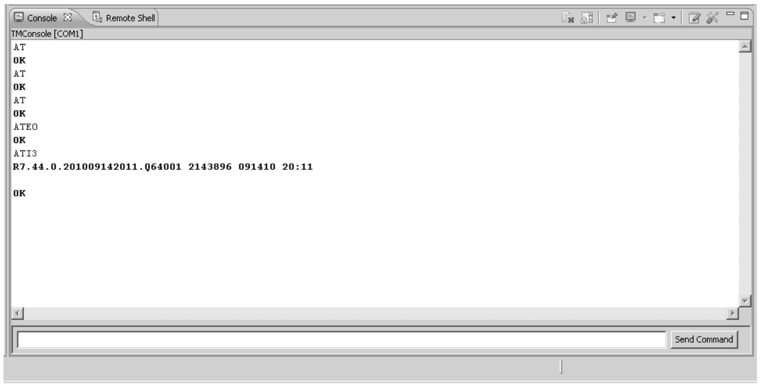
	Flows	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	ADL																
▣	ATI	✓														✓	
	AUD																
	CUS1																
	CUS2																
	CUS7																
	DATA																
▣	DEV	✓		✓		✓	✓	✓	✓			✓	✓				
▣	FCM								✓								
	FLASH																
	GRR																

	Flows	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
	ADL																
▣	ATI																
	AUD																
	CUS1																
	CUS2																
	CUS7																
	DATA																
▣	DEV																
▣	FCM																
	FLASH																
	GRR																

Figure 44 – Traces configuration

9. The traces logged by the Open AT Application will be displayed in the Traces window based on the Trace levels selected.

10. Console window can be viewed using Window->Show View->Other. In Show View window select General->Console and then click OK. The Console enables the user to enter AT commands and receive the responses for the same.



```
AT
OK
AT
OK
AT
OK
ATE0
OK
ATI3
R7.44.0.201009142011.Q64001 2143896 091410 20:11

OK
```

Figure 45 – Console

2.2. Open AT Perspective

This perspective is used to develop Open AT Applications using the ADL APIs and to create Run/Debug configurations for Open AT Application execution. The Open AT application can be executed in either Target mode or Remote/Debug mode of execution. The execution mode is decided based on the selected tool chain (Target or RTE) and the created Run Configuration.

2.3. Debug Perspective

This perspective is used for debugging an Open AT Application in Remote Debug mode. The perspective is automatically called once a breakpoint in the Open AT Application is hit. The perspective allows doing a step by step debugging of an Open AT Application. The steps involved in debugging have been discussed in details in chapter 5.

3. Open AT Application

The Open AT application can be compiled using the following compilers:

- ARM ELF GCC
- ARM EABI GCC
- RVDS
- MINGW

3.1. ARM ELF GCC

The GNU Compiler Collection (usually shortened to GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is often the compiler of choice for developing software that is required to execute on a wide variety of hardware and/or operating systems.

3.2. ARM EABI GCC

This toolchain requires some internal functions at the link stage as soon as C++ is used. These functions are provided in a stub, which has to be added to every Open AT Application project compiled with the ARM EABI GCC toolchain and which embeds C++ code. This compiler is selected by default, while creating an Open AT Project in Developer Studio for OASIS 2.34 and above. List of other compilers will also be available, the user can select all other compilers also.

3.3. RVDS

The ARM RealView Development Suite 4.0 Professional is a complete, end-to-end solution for software development supporting all ARM processors and ARM CoreSight™ debug technology. This product enables developers to begin software development, optimization and test ahead of silicon availability, significantly reducing application time-to-market and ensuring the highest degree of software quality. RealView Development Suite 4.0 Professional incorporates the best-in-class ARM Compiler and market-leading ARM Profiler enabling applications to easily achieve both high performance and optimal code size.

Supported Platforms
RVDS 4.0 supports the following platforms and service packs:

- Windows XP Professional service pack 2
- Windows Vista Business service pack 1
- Windows Vista Enterprise service pack 1
- Windows Server 2003 (Compiler only)
- Red Hat Enterprise Linux WS version 4 for x86 using GNOME Window Manager and Bash Shell
- Red Hat Enterprise Linux WS version 5 for x86 using GNOME Window Manager and Bash Shell

3.4. MINGW

MinGW is a native Windows port of the GNU Compiler Collection (GCC), with freely distributable import libraries and header files for building native Windows applications. MinGW is used to compile the Open AT Application in remote i.e. RTE mode. It is actually integrated into the IDE itself and it uses the Windows C runtime libraries.

3.5. Comparison between the types of compilers

GCC	ADS	RVDS
The generated binary file size is relatively of larger size	The generated binary file size is relatively smaller than GCC but larger when compared to RVDS	The generated binary file size is of minimal size compared to GCC and ADS
It uses –ggdb3 which produces debugging information specifically intended for gdb	It uses –g debug option which produces debugging information in the OS's native format	It uses –g debug option which produces debugging information in the OS's native format
Installation is free	Installation requires license	Installation requires license
Execution time required is more compared to ADS	Execution time required is less compared to GCC and RVDS	Execution time required is more compared to ADS and GCC
Compilation time is more	Compilation time is less compared to GCC	Compilation time is least
The qualifier used for structure/union alignment on single byte boundary is __attribute__((__packed__))	The qualifier used for structure/union alignment on single byte boundary is __packed__	The qualifier used for structure/union alignment on single byte boundary is __packed__

3.6. Limitations of Compilers

In case of ARM ELF GCC compiler, usage of "%f" mode in the wm_sprintf function will lead to an ARM exception (product reset).

Maximum optimization level possible is –O3 in ARM ELF GCC and RVDS compilers.

4. Open AT Application Execution

4.1. Target Application Execution

In this mode, the Open AT Application binary is downloaded in the embedded module (programmed in the flash memory) and then executed. The Open AT Application uses the resources of the embedded module to perform the required operation. In this mode, the Open AT Application should be compiled using ARM ADS, RVDS compiler or the GCC compiler. This mode is used when a standalone application has to be deployed in the embedded module. The Open AT Application is self-reliant and can run without user intervention. The successful compilation and linking of the Open AT Application generates a .dwl file.

The generated binary file (.dwl file) is downloaded to the embedded module by Developer Studio when a Target Run Configuration is created in the Open AT perspective. After the download is complete, the embedded module is restarted using the AT+CFUN=1 command. The newly created Open AT Application is then started using the AT+WOPEN=1 command. All these commands are issued by the Developer Studio and the user does not have to do anything.

The AT+WOPEN=0 command can be used to stop the embedded Open AT Application. The AT+WOPEN=4 command can be used to erase the embedded Open AT Application.

4.2. Remote Application Execution

In the remote mode, the Open AT Application is executed on the development environment (PC) and uses its own resources such as the memory and clock. This mode, as the name indicates, is used to debug Open AT Application.

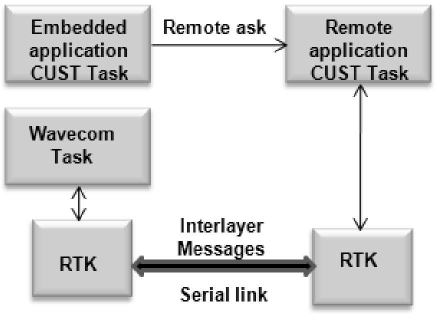


Figure 46 - Remote Task Environment

The Open AT Application runs using the Real Time Execution Environment (RTE) which is integrated with Developer Studio. This mode can be used to debug the Open AT Application by inserting various breakpoints and traces. Moreover, dynamic changes in the contents of the memory can also be seen using the tools available in Developer Studio. This mode can be used during the development of the Open AT Application before releasing the product (which will be downloaded to the embedded module).

The RTE is used to send the instructions directly to the embedded module, which then executes these instructions and performs the required operation. The Remote Task Execution (Remote Mode) image shows a screenshot of the Remote Task Monitor which is displayed when the Open AT Application is running in the remote mode.

Before starting the remote Open AT Application, set the appropriate ADL level traces from the Remote Task Monitor.

5. Introduction to GDB

The GNU Debugger, usually called just GDB, is the standard debugger for the GNU software system. GDB, the GNU Project debugger, allows you to see what is going on 'inside' a program while it executes or what a program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behaviour.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

The program being debugged can be written in Ada, C, C++, Objective-C, Pascal (and many other languages). Those programs might be executing on the same machine as GDB (native) or on another machine (remote). GDB can run on most popular UNIX and Microsoft Windows variants.

6. Introduction to JTAG

JTAG (Joint Test Action Group) is an in-circuit debugging solution capable of tracking down issues which may occur in routines but cannot be reproduced with the Remote Tasks Environment. Using a compliant debugger and the proper configuration, Open AT customers can step through the code of their Open AT application on the AirPrime WMP Series.

6.1. Need for JTAG

JTAG was originally designed for printed circuit boards, but is primarily used for accessing sub-blocks of integrated circuits, and is also useful as a mechanism for debugging embedded systems, providing a convenient "back door" into the system. When used as a debugging tool, an in-circuit emulator - which in turn uses JTAG as the transport mechanism - enables a programmer to access an on-chip debug module which is integrated into the CPU, via the JTAG interface. The debug module enables the programmer to debug the software of an embedded system.

Besides debugging, the second purpose of the JTAG interface is allowing device programmer hardware to transfer data into internal non-volatile device memory. Some device programmers serve a double purpose for programming as well as debugging the device.

6.2. JTAG in Sierra board

The following figure is a typical schematic for use of the Sierra Wireless debug interface on the AirPrime WMP100.

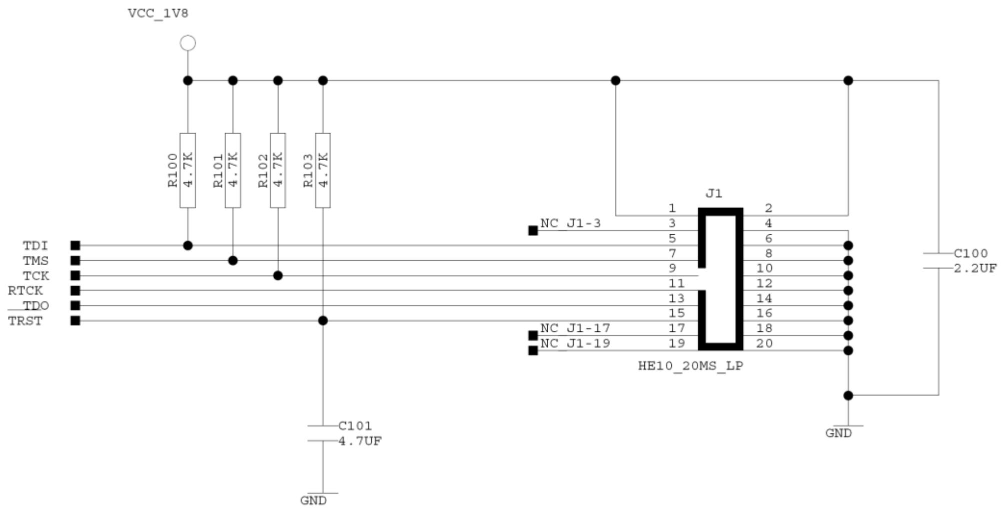
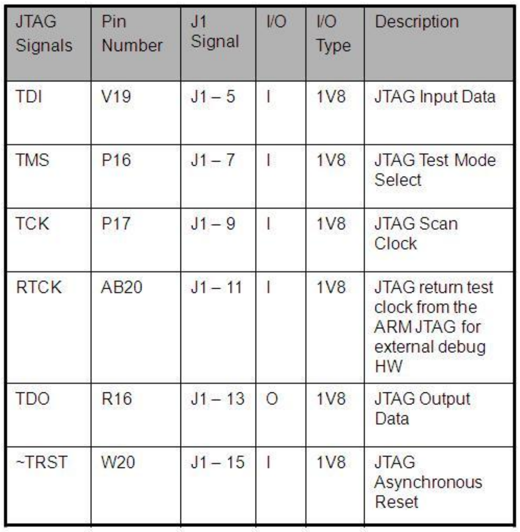
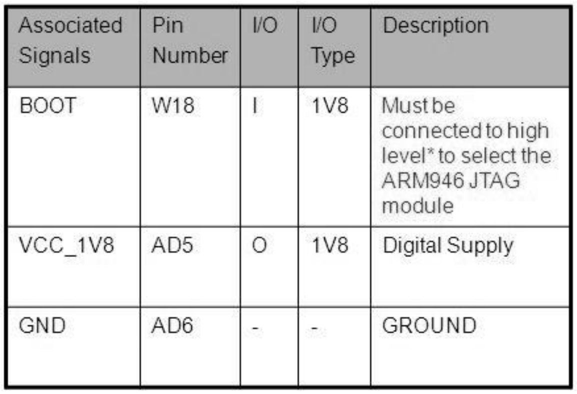


Figure 47: Schematic for use of the Sierra Wireless debug interface on the AirPrime WMP100.



JTAG Signals	Pin Number	J1 Signal	I/O	I/O Type	Description
TDI	V19	J1 – 5	I	1V8	JTAG Input Data
TMS	P16	J1 – 7	I	1V8	JTAG Test Mode Select
TCK	P17	J1 – 9	I	1V8	JTAG Scan Clock
RTCK	AB20	J1 – 11	I	1V8	JTAG return test clock from the ARM JTAG for external debug HW
TDO	R16	J1 – 13	O	1V8	JTAG Output Data
~TRST	W20	J1 – 15	I	1V8	JTAG Asynchronous Reset



Associated Signals	Pin Number	I/O	I/O Type	Description
BOOT	W18	I	1V8	Must be connected to high level* to select the ARM946 JTAG module
VCC_1V8	AD5	O	1V8	Digital Supply
GND	AD6	-	-	GROUND

* Refer to the APN WMP_Family_JTAG_Debug_Application_Note-001_002.pdf for the BOOT signal configuration example.

6.3. AirPrime WMP Series Settings

The following subsection details how to set up the AirPrime WMP Series in order to begin the debugging process.

Please refer to the AT commands Interface Guide for complete information regarding the commands in the following subsections.

6.3.1. Enabling JTAG Debug mode: +WCFM

Use the +WCFM command to verify that the JTAG Debug Mode is currently enabled on the AirPrime WMP Series. Verify that the second parameter of the command's answer displays that it is enabled by executing command AT+WCFM=2. If the JTAG Debug Mode is not currently enabled, issue the AT+WCFM command to enable it.

Notes
The JTAG debug mode can be enabled with <mode>= 1 and the parameter <FtrMask>= 80 (Sierra debug feature). The +WCFM command requires a password to enable JTAG Debug mode (and other modes as well). Please contact your Sierra FAE if you do not readily have your password.

6.3.2. Activation JTAG Debug Mode: +WDEBUG

Activate the JTAG Debug mode by issuing the +WDEBUG command as follows:

```
AT+WDEBUG=1
```

Verify that the JTAG Debug mode is properly activated using the AT+WDEBUG? command and that an error command was not issued. The response confirming that JTAG Debug mode is active is as follows:

```
+WDEBUG:1,1

OK
```

6.3.3. Resetting the AirPrime WMP Series

Reset the AirPrime WMP Series to complete setting up JTAG Debug mode. You can reset it via a soft reset by issuing the +CFUN command as follows: AT:

```
AT+CFUN=1
```

Otherwise you can issue a hard reset by using the switch on the hardware itself. Please refer to the related documentation if you are unsure how to perform a hard reset.

NOTES
When the +WDEBUG read command is issued after resetting the AirPrime WMP Series, it will display the following response if the JTAG Debug mode is properly activated:
+WDEBUG:1,0
OK
For more details on how to use this JTAG interface, please refer to WMP_Family_JTAG_Debug_Application_Note-001_002.pdf.

7. Functioning Modes

Open AT Application can work in any one of the following functioning modes:

- Standalone External Application
- Embedded Application in Standalone Mode
- Cooperative Mode

7.1. Standalone External Application

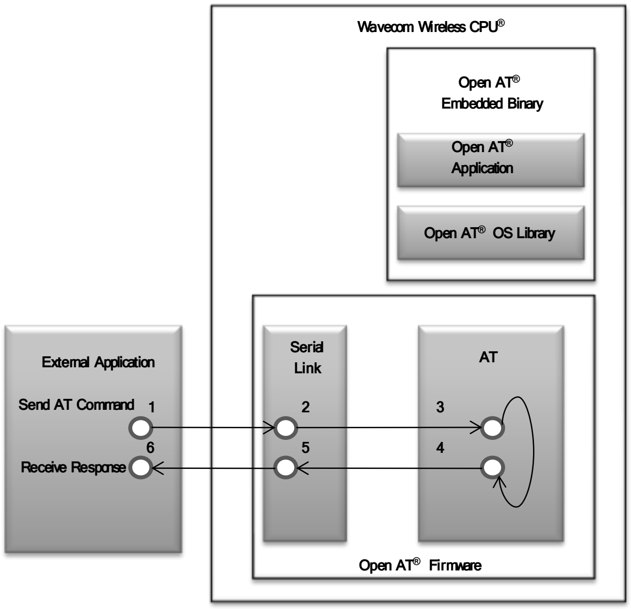


Figure 48 - Standalone External Application

In this mode of operation the commands are sent by the external application to the embedded module that you want to process, and the response is sent back to the external application.

The Standalone External Application works in the following steps:

1. The external application sends an AT command.
2. The serial link transfers the command to the Firmware.
3. The Firmware function processes the command.
4. After processing the command, it sends back the response to external application.
5. The response is sent through the serial link.
6. The external application receives the response.

7.2. Embedded Application in Standalone Mode

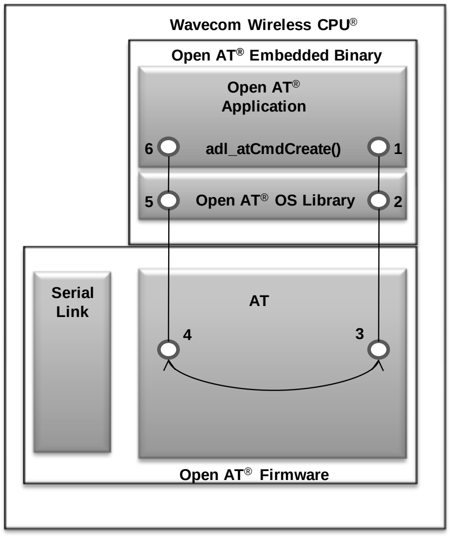


Figure 49 - Embedded Application in Standalone Mode

In this mode of operation the Open AT Application is downloaded over the embedded module and is run in the target mode. The Open AT Application and the embedded module do not require any external interface to drive the embedded module. The Open AT Application will drive the product by sending the command and handling the response with the embedded module.

The embedded application in standalone mode works in the following steps:

1. The embedded Open AT Application calls the adl_atCmdCreate function to send an AT command.
2. The Open AT OS calls the appropriate AT command from the Firmware.
3. The AT function processes the command.
4. The AT function sends the AT response to the embedded Open AT Application.
5. This response ID is dispatched by the Open AT OS which calls the event handler associated with adl_atCmdCreate function of the Open AT Application.
6. The event handler processes the response.

7.3. Cooperative Mode

In the cooperative mode of operation the embedded Open AT Application and external application interact with each other. In this mode of operation, the embedded Open AT Application spy on all the commands coming from the external application to the embedded module via the serial link.

In this case, various levels of subscription are available for the command and responses, depending on the requirement.

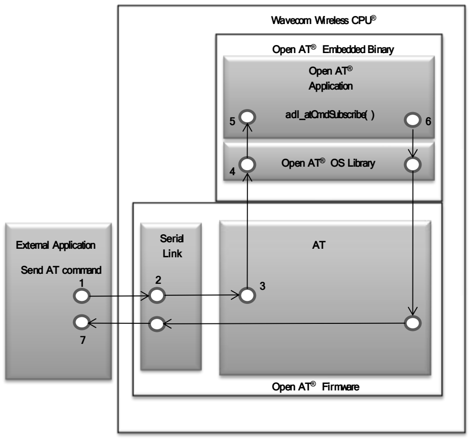


Figure 50 - Cooperative Mode

Follow the steps below to pre-parse a subscription:

1. The embedded Open AT Application subscribes to the command by calling the adl_atCmdSubscribe function.
2. The Open AT OS calls the appropriate function from the Firmware.
3. The AT function sets the subscription.

Follow the steps below to process an AT command:

1. The external application sends an AT command.
2. The serial link transfers the AT command to the AT processor function in the Firmware.
3. The AT function transmits it to the Open AT Application without processing it.
4. This response ID is dispatched by the Open AT OS which calls the event handler associated with adl_atCmdSubscribe function of the Open AT Application.
5. The event handler processes the response.

Summary

The following points have been covered in this section

- Developer Studio provides different perspectives that can be used for different purpose:
  - Open AT perspective to develop Open AT Application.
  - Target Management perspective to communicate with the embedded module.
  - Debug perspective to do a step by step debugging of Open AT Application.
- Application execution modes:
  - Remote Mode: Open AT Application utilizes the resources of development environment (PC) and utilizes its resources (like memory and clock). Suitable for debugging purposes.
  - Target Mode: Application actually downloaded in the embedded module and executed. Utilizes the embedded module resources.

CHAPTER 7

AT Commands and Responses

1. Objective

The objective of this chapter is to help you build familiarity with the APIs that can be used to subscribe/unsubscribe to the AT Commands and Responses. Open AT allows you to build and subscribe to custom (user-defined) commands. You can define the functionality for these custom commands to enhance the usability of the Open AT Application.

This chapter provides the following information on AT commands and responses:

- What are responses?
- What are the various types of responses?
- How can you subscribe to responses?
- How can you unsubscribe to responses?
- How to send responses from Open AT Applications?
- How to create and execute a standard AT command from the Open AT Application?
- How to subscribe to AT commands and define their functionality?

2. Responses

All the AT commands and response-related APIs are declared in the adl_at.h header file. This file must be included in the Open AT Application to use these APIs.

The embedded module executes the AT commands and generates possible responses. The response generated depends on the type of command executed.

The possible responses that that can be received are:

- OK: Indicates the successful execution of the command.
- Indications: Unsolicited/intermediate indications received either from the embedded module or from the network.
- Error: To indicate failure in executing the command.

Errors generated during the execution of these AT commands could be due to the following reasons:

- Incorrect syntax/parameters of the AT command
- Network-related errors
- SIM/hardware-related errors

In case of an error, the string ERROR or +CME ERROR: <error_no> responses are displayed.

The responses generated provide feedback about the success or failure of AT command execution, and also indicate the occurrence of any new events.

3. Classification of Responses

There are three categories of responses:

- Terminal Response
- Unsolicited Response
- Intermediate Response

3.1. Terminal Response

Terminal responses are the final outcome of an AT command, and indicate the success or failure of the command executed. Examples of this category are OK, +CMS ERROR: <error_no> and +CME ERROR: <error_no>.

3.2. Unsolicited Response

Unsolicited responses are sent without being requested for, and indicate occurrence of events like network registration, new SMS or incoming call. These responses are not related to the execution of any command.

3.3. Intermediate Response

Intermediate responses indicate the progress of the execution of certain commands. For example, to send an SMS, use the AT+CMGS command. After executing the AT+CMGS=”<dest_no>” command, an intermediate response (“>”) is sent by the embedded module. This response prompts the entry of SMS text suffixed by a Ctrl+Z character.

The Open AT Application can only subscribe to unsolicited responses, and not to terminal and intermediate responses.

4. Subscribing to Unsolicited Responses

Open AT OS provides the following API to subscribe to unsolicited responses.

Prototype:

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

The API associates a call-back (handler) function to a specific unsolicited response. The call-back function is executed whenever the unsolicited event is received. This function will contain your specific implementation.

Parameters:

UnSostr: This parameter is the name of the unsolicited response to which you want to subscribe. For instance, this parameter can contain the string "+WIND: 4". It means that you are subscribing to the "+WIND: 4" unsolicited event.

There is a predefined set of response IDs which can be used by programmers. These responses include the ADL_STR_NO_STRING, ADL_STR_OK values and others listed in the following table:

Value	Response Type
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 responses are before this line.
ADL_STR_RING	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_CRC	+CRC:
ADL_STR_CGEREP	+CGEREP:
ADL_STR_LAST	Last string ID.

UnSohdl: The call-back function that will be called on receipt of the unsolicited event specified in UnSostr (first parameter). This function should be declared as follows:

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

The call-back handler is invoked and information such as the Response ID or actual string response is passed. The UnSohdl callback function must return TRUE if you want the unsolicited responses to be sent to the external application (Terminal Emulator/HyperTerminal).

If the UnSohdl callback function returns FALSE, then the unsolicited response will not be sent to the external application.

The parameter of this callback function is a structure of the adl_atUnsolicited_t type. This structure will hold the unsolicited response to which you have subscribed. The structure is declared as follows:

```
        typedef struct
```

```
{
adl_strID_e RspID;
adl_atPort_e Dest;
u16 StrLength;
ascii StrData[1];
} adl_atUnsolicited_t;
```

If the response is a terminal response (member of adl_rsp_ID_e), then RspID contains the response ID.

The Dest member indicates the destination port to which unsolicited response will be sent.

The StrLength member contains the length of the response.

The StrData is the pointer to the received response.

Returned values:

OK is returned if the unsolicited subscription was successful.

ERROR is returned if there was an error in subscribing to the unsolicited response.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTES
The call-back function will be executed every time the unsolicited response is received. You should explicitly unsubscribe from the unsolicited response to prevent the call-back function from being executed each time the response is received.
An unsolicited response can be subscribed to several times in your Open AT Application. If you subscribe for an unsolicited response and provide func1 () as the call-back function and then again subscribe for the same response by providing func2 () as the call-back, both the functions (func1 () and func2 ()) will be executed on receipt of that unsolicited response.
The Open AT Application can only subscribe to unsolicited responses, and not to terminal and intermediate responses.

5. Unsubscribing from Responses

Open AT OS provides following API to unsubscribe from unsolicited responses.

Prototype:

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

Parameters:

UnSostr: This parameter represents the unsolicited string from which you want to unsubscribe. This should be the same string that was subscribed to using the adl_atUnSoSubscribe() API. Refer to the enumeration adl_rspID_e for the values which can be assigned to this string.

UnSohdl: This parameter indicates the call-back function that was associated with the subscribed unsolicited response. This is the same function that was specified in the adl_atUnSoSubscribe () API. Refer to the section on
UnSohdl for the description of this call-back function.

Returned values:

OK is returned if the unsolicited response is unsubscribed successfully.

ERROR is returned if there is an error in unsubscribing from the unsolicited response.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTES
You can only unsubscribe from unsolicited responses to which you had subscribed.
After unsubscribing, the call-back function will not be called on receipt of same event.

6. Sample Code

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

bool wind4_handler (adl_atUnsolicited_t *paras)
{
  TRACE((1,"Inside wind4 handler"));
  TRACE((1, "\r\n+WIND:4 received"));
  adl_atUnSoUnSubscribe("+WIND: 4",wind4_handler);
  return FALSE;
}
void adl_main ( adl_InitType_e InitType )
{
  TRACE (( 1, "Embedded Application: Main" ));
  if (adl_atUnSoSubscribe("+WIND: 4",wind4_handler)!=OK)
    TRACE((1,"Error in subscribing +WIND: 4"));
  else
    TRACE((1,"Successfully subscribed +WIND: 4"));
}
```

7. Sending Responses from the Open AT Application

Open AT OS provides APIs using which you can send standard responses to the external application from within the Open AT Application. In this way, you can create custom responses. The APIs which can be used to send responses are:

adl_atSendResponse() API
adl_atSendResponseSpe() API
adl_atSendStdResponse() API
adl_atSendStdResponseSpe() API
adl_atSendStdResponseExt() API

adl_atSendStdResponseExtSpe() API

adl_atSendStdResponseExtStr() API

adl_atSendUnsoResponse() API

adl_atSendResponsePort() API

adl_atSendStdResponsePort API

adl_atSendStdResponseExtPort API

7.1. adl_atSendResponse API

This is the most commonly used API which allows you to send any type of responses from within your Open AT Application to the external application. This API can be used to send text as various kinds of responses to the external application. This gives you the flexibility to send the text as either a terminal, unsolicited or intermediate response. The Type parameter determines the type of the response associated to the text string.

Prototype:

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

Parameters:

Type: This parameter determines the type of the response which is associated to the text string that you want to send and the port to which the string needs to be sent. This is done using the ADL_AT_PORT_TYPE macro.

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

Parameters:

_port: This parameter is of adl_atPort_e type which specifies the port on which the text string should be sent. Note that the port should be opened otherwise the response will not be displayed.

```
typedef adl_port_e  adl_atPort_e;
typedef enum
{
        ADL_PORT_NONE,
        ADL_PORT_UART1,
        ADL_PORT_UART2,
        ADL_PORT_USB,

         ADL_PORT_MAX = ADL_PORT_USB,

        ADL_PORT_UART1_VIRTUAL_BASE = 0x10, // Base for UART1 virtual
                                            // ports
         ADL_PORT_UART2_VIRTUAL_BASE = 0x20, // Base for UART2 virtual
 // ports
```

```
                ADL_PORT_USB_VIRTUAL_BASE  = 0x30, // Base for USB virtual ports
               ADL_PORT_BLUETOOTH_VIRTUAL_BASE = 0x40, // Base for BlueTooth
// virtual ports
               ADL_PORT_GSM_BASE = 0x50, // GSM CSD call data port
               ADL_PORT_GPRS_BASE  = 0x60, // GPRS session port
                ADL_PORT_OPEN_AT_VIRTUAL_BASE  = 0x80 // Base for Open-AT
    // application tasks
} adl_port_e;
```

_type: The values this parameter can take are:

- ADL_AT_RSP: This value indicates that the text should be sent as a final response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used. When this type of responses is sent, all the buffered unsolicited responses will be flushed on the required port.
- ADL_AT_UNS: This value indicates that the text should be sent as an unsolicited response. These responses will be sent to all the opened ports if ADL_AT_PORT_TYPE macro is not used. This type of responses is buffered automatically till the terminal response is sent.
- ADL_AT_INT: This value indicates that the text should be sent as an intermediate response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used.

String: This parameter is a pointer to the string which should be sent to the external application as a response. The type of the response string depends on the Type parameter of the adl_atSendResponse() API.

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7.2. adl_atSendResponseSpe API

This API can be used to send specified text as various kinds of responses to the external application. This gives you the flexibility to send the text as either a terminal, unsolicited or intermediate response. The Type parameter determines the type of the response associated to the text. With the NI provided, the associated command is found. If the command had subscribed to this response, then the response handler is called. Otherwise, the response is sent to the port provided.

NOTES
NI parameter is to hold the Notification Identifier provided by the command initiating the response. This parameter is helpful in identifying the command that subscribed for the response in the case of multiple subscriptions.
This parameter is related to the adl_atCmdSendExt() API and hence please refer to the Note after 7.8.4 and section 7.8.5 Sample Code to understand the usage of the notification identifier (NI).

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)

Parameters:

_port: This parameter is of adl_atPort_e type which specifies the port on which the text string should be sent. Note that the port should be opened otherwise the response will not be displayed.

_type: The values this parameter can take are:

- ADL_AT_RSP: This value indicates that the text should be sent as a final response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used. When this type of responses is sent, all the buffered unsolicited responses will be flushed on the required port.
- ADL_AT_UNS: This value indicates that the text should be sent as an unsolicited response. These responses will be sent to all the opened ports if ADL_AT_PORT_TYPE macro is not used. This type of responses is buffered automatically till the terminal response is sent.
- ADL_AT_INT: This value indicates that the text should be sent as an intermediate response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used.

Text: The string of response.

NI: Notification identifier to find the associate command.

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7.3. adl_atSendStdResponse API

This API can be used to send terminal responses to the external application. This API gives you the facility to send only the standard responses defined as the elements of the adl_strID_e enumeration to the external application. The Type parameter defines the type that must be associated with the response ID.

Prototype:

s32 adl_atSendStdResponse (u8 Type, adl_strID_e RspID)

Parameters:

Type: This parameter determines the type of the response which is associated to the text string that you want to send and the port to which the string needs to be sent. This is done using the ADL_AT_PORT_TYPE macro.

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

Parameters:

_port: This parameter is of adl_atPort_e type which specifies the port on which the text string should be sent. Note that the port should be opened otherwise the response will not be displayed.

_type: The values this parameter can take are:

- ADL_AT_RSP: This value indicates that the text should be sent as a final response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used. When this type of responses is sent, all the buffered unsolicited responses will be flushed on the required port.
- ADL_AT_UNS: This value indicates that the text should be sent as an unsolicited response. These responses will be sent to all the opened ports if ADL_AT_PORT_TYPE macro is not used. This type of responses is buffered automatically till the terminal response is sent.
- ADL_AT_INT: This value indicates that the text should be sent as an intermediate response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used.

RspID: This parameter indicates the response string that must be sent.

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7.4. adl_atSendStdResponseSpe API

This API can be used to send terminal responses to the external application. This API gives you the facility to send only the standard responses defined as the elements of the adl_strID_e enumeration to the external application. The Type parameter defines the type that must be associated with the response ID. 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 )
```

Parameters:

_port: This parameter is of adl_atPort_e type which specifies the port on which the text string should be sent. Note that the port should be opened otherwise the response will not be displayed.

_type: The values this parameter can take are:

- ADL_AT_RSP: This value indicates that the text should be sent as a final response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used. When this type of responses is sent, all the buffered unsolicited responses will be flushed on the required port.
- ADL_AT_UNS: This value indicates that the text should be sent as an unsolicited response. These responses will be sent to all the opened ports if ADL_AT_PORT_TYPE macro is not used. This type of responses is buffered automatically till the terminal response is sent.
- ADL_AT_INT: This value indicates that the text should be sent as an intermediate response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used.

RspID: The ID of the response.

NI: Notification identifier to find the associate command.

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7.5. adl_atSendStdResponseExt API

This API is an extension to the previous adl_atSendStdResponse() API. This is because it allows you to send arguments along with various terminal responses. For example, using adl_atSendStdResponse() API, you can send the +WIND: response (as defined in the adl_rspID_e enumeration). However, to send a +WIND: 4 response or +WIND: 7 response (that is, the +WIND response with the argument), you must use the adl_atSendStdResponseExt() API.

Prototype:

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

Parameters:

Type: This parameter determines the type of the response which is associated to the text string that you want to send and the port to which the string needs to be sent. This is done using the ADL_AT_PORT_TYPE macro.

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

Parameters:

_port: This parameter is of adl_atPort_e type which specifies the port on which the text string should be sent. Note that the port should be opened otherwise the response will not be displayed.

_type: The values this parameter can take are:

ADL_AT_RSP: This value indicates that the text should be sent as a final response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used. When this type of responses is sent, all the buffered unsolicited responses will be flushed on the required port.
ADL_AT_UNS: This value indicates that the text should be sent as an unsolicited response. These responses will be sent to all the opened ports if ADL_AT_PORT_TYPE macro is not used. This type of responses is buffered automatically till the terminal response is sent.
ADL_AT_INT: This value indicates that the text should be sent as an intermediate response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used.

RspID: This parameter indicates the response ID that must be sent.

arg: This parameter represents the response argument. According to the RspID, this argument should be an u32 integer, or an ascii * string. For example, an integer value should be assigned to this argument when it is used with an RspID "ADL_STR_WIND". For "ascii *" string, the pointer to the string should be type cast to u32 and passed as a last parameter.

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7.6. adl_atSendStdResponseExtSpe API

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 )
```

Parameters:

_port: This parameter is of adl_atPort_e type which specifies the port on which the text string should be sent. Note that the port should be opened otherwise the response will not be displayed.

_type: The values this parameter can take are:

ADL_AT_RSP: This value indicates that the text should be sent as a final response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used. When this type of responses is sent, all the buffered unsolicited responses will be flushed on the required port.
ADL_AT_UNS: This value indicates that the text should be sent as an unsolicited response. These responses will be sent to all the opened ports if ADL_AT_PORT_TYPE macro is not used. This type of responses is buffered automatically till the terminal response is sent.
ADL_AT_INT: This value indicates that the text should be sent as an intermediate response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used.

RspID: The ID of the response.

arg: This parameter represents the response argument. According to the RspID, this argument should be an u32 integer, or an ascii * string. For example, an integer value should be assigned to this argument when it is used with an RspID "ADL_STR_WIND". For "ascii *" string, the pointer to the string should be type cast to u32 and passed as a last parameter.

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7.7. adl_atSendStdResponseExtStr API

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 (u16 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 )
```

Parameters:

_port: This parameter is of adl_atPort_e type which specifies the port on which the text string should be sent. Note that the port should be opened otherwise the response will not be displayed.

_type: The values this parameter can take are:

ADL_AT_RSP: This value indicates that the text should be sent as a final response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used. When this type of responses is sent, all the buffered unsolicited responses will be flushed on the required port.

ADL_AT_UNS: This value indicates that the text should be sent as an unsolicited response. These responses will be sent to all the opened ports if ADL_AT_PORT_TYPE macro is not used. This type of responses is buffered automatically till the terminal response is sent.
ADL_AT_INT: This value indicates that the text should be sent as an intermediate response string. These responses will be sent to UART1 if ADL_AT_PORT_TYPE macro is not used.

RspID: The ID of the response.

arg: This parameter represents the response argument. According to the RspID, this argument should be an u32 integer, or an ascii * string. For example, an integer value should be assigned to this argument when it is used with an RspID "ADL_STR_WIND". For "ascii *" string, the pointer to the string should be type cast to u32 and passed as a last parameter.

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7.8. adl_atSendUnsoResponse API

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_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.
RIpulse:	This RI pulse flag, if TRUE, RI signal is pulsed.

NOTES
The RI pulse generation behaviour depends on "+WRIM" AT command parameter:

- *If mode parameter of "+WRIM" AT command is set to 0, RI signal cannotbe pulsed by adl_atSendUnsoResponse.*
- *RI pulse duration depends on pulse_width parameter of "+WRIM" ATcommand.*

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7.9. adl_atSendResponsePort () macro

This macro is used to send a custom response string to a specific port. The response string can be of the following types:

- Unsolicited
- Terminal
- Intermediate

Prototype:

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

Parameters:

_t: This parameter determines the type of the response which is associated to the text string that you want to send. The values this parameter can take are:

- ADL_AT_RSP: This value indicates that the text should be sent as a final response string.
- ADL_AT_UNS: This value indicates that the text should be sent as an unsolicited response.
- ADL_AT_INT: This value indicates that the text should be sent as an intermediate response string.

_p: This parameter is of adl_atPort_e type which specifies the port on which the text string should be sent. Note that the port should be opened otherwise the response will not be displayed.

_r: This parameter is a pointer to the string which is sent to the specified port as a response. The type of the response string depends on the _t parameter.

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7.10. adl_atSendStdResponsePort() macro

This macro can be used to send standard defined responses such as OK, ERROR (unsolicited/terminal/intermediate) to a specific port.

Prototype:

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

Parameters:

_t: This parameter determines the type of the response which is associated to the text string that you want to send. The values this parameter can take are:

- ADL_AT_RSP: This value indicates that the text should be sent as a final response string.
- ADL_AT_UNS: This value indicates that the text should be sent as an unsolicited response.
- ADL_AT_INT: This value indicates that the text should be sent as an intermediate response string.

_p: This parameter is of adl_atPort_e type which specifies the port on which the text string should be sent. Note that the port should be opened otherwise the response will not be displayed.

_r: This parameter is a pointer to the string which is sent to the specified port as a response. The type of the response string depends on the _t parameter.

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7.11. adl_atSendStdResponseExtPort () macro

This macro can be used to send a standard defined response with a parameter (unsolicited/terminal/intermediate) on a specific port. For e.g. '+CME ERROR: 10" where "+CME ERROR" is a standard defined response and "10" is a parameter for the response.

Prototype:

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

Parameters:

_t: This parameter determines the type of the response which is associated to the text string that you want to send. The values this parameter can take are:

- ADL_AT_RSP: This value indicates that the text should be sent as a final response string.
- ADL_AT_UNS: This value indicates that the text should be sent as an unsolicited response.
- ADL_AT_INT: This value indicates that the text should be sent as an intermediate response string.

_p: This parameter is of adl_atPort_e type which specifies the port on which the text string should be sent. Note that the port should be opened otherwise the response will not be displayed.
_r: This parameter indicates the response ID that must be sent.

_a: This parameter represents the response argument.

Returned values:

OK is returned if the response is sent successfully.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTES
adl_atSendResponseSpe, adl_atSendStdResponseSpe, adl_atSendStdResponseExtSpe are to be used with adl_atCmdSendExt function.
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).

7.12. Sample Code

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

void adl_main ( adl_InitType_e InitType )
{
adl_atSendResponse(ADL_AT_UNS,"\r\nUnsolicited Responses\r\n");
adl_atSendStdResponse(ADL_AT_UNS,ADL_STR_BUSY);
adl_atSendStdResponseExt(ADL_AT_UNS,ADL_STR_WIND,4);
adl_atSendResponse(ADL_AT_INT,"\r\nIntermediate response\r\n");
adl_atSendStdResponse(ADL_AT_INT,ADL_STR_BUSY);
adl_atSendStdResponseExt(ADL_AT_INT,ADL_STR_WIND,20);
adl_atSendResponse(ADL_AT_RSP,"\r\nFinal Responses\r\n");
adl_atSendStdResponse(ADL_AT_RSP,ADL_STR_BUSY);
adl_atSendStdResponseExt(ADL_AT_RSP,ADL_STR_WIND,4);
adl_atSendResponsePort (ADL_AT_RSP, ADL_PORT_UART1, ”\r\nRsp to a port”);
}
```

8. Sending Standard AT Commands from the Open AT Application

Open AT OS provides APIs using which you can send standard AT commands from the Open AT Application. The APIs which can be used for this are:

adl_atCmdCreate () API
adl_atCmdSend () API
adl_atCmdSendExt () API
adl_atCmdSendText() API

8.1. adl_atCmdCreate API

The following API allows you to send standard AT commands. The response of the executed command can be captured in the associated callback function. You can also subscribe to various responses of the command to be sent.

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

Parameters:

Cmdstr: This parameter contains the command that you want to send (or execute). The command should be an AT command. Refer to the AT command interface guide for the various commands and their formats.

Rspflag: This parameter determines whether or not the response and the intermediate response of the command being created should be sent to the external application. It can have the following values:

TRUE: This indicates that the responses and the intermediate responses of the created command (that are not subscribed) will be sent to the external application.

FALSE: This indicates that the responses and intermediate responses of the created command (that are not subscribed) will not be sent to the external application.

This parameter can also be used to determine the port from which the command is to be created. For instance, if you want a particular response to be created in such a way that it is emulated to be created from UART1, then the RspFlag parameter can be used to define this behaviour. This is done using the macro ADL_AT_PORT_TYPE. For instance,

ADL_AT_PORT_TYPE(ADL_PORT_UART1,TRUE) would emulate as if the command is created from UART1. This is useful in situations, when you want to change the parameters for a particular port (for e.g. baud rate, character framing for a particular port).

Rsphdl: This parameter is the callback function associated with all the subscribed responses and intermediate responses of the command being created and sent. This function will be called whenever the subscribed responses of the command are received by the Open AT Application. This function has the following prototype:

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

The argument of this function is of the adl_atResponse_t structure. This will hold the response to which you have subscribed. This structure has the following definition:

```
typedef struct
{
 adl_strID_e RspID; // Response ID // RspID
 adl_port_e Dest;     // Dest
 u16 StrLength;      //Length of unsolicited  // 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];  //Unsolicited response.  // 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.

< … > This parameter represents a list of standard and intermediate responses to which you want to subscribe. When these responses are received, the callback function (Rsphdl) will be executed. To mark the end of the list, the last element of this list should be NULL. If this list contains only two elements ("*" and NULL), then all the responses of the command being sent will result in the invocation of the callback function.

Returned values:

OK is returned if the response is sent successfully.

ERROR is returned in case of an error.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTES
You can specify which intermediate or terminal responses of the command being created should lead to the execution of the callback function (Rsphdl). Specify the responses in the (…) variable list parameter of adl_atCmdCreate () API.
If the variable list parameter (…) contains "" and NULL, then any of the responses which the command will generate result in the execution of the callback function.*

You also have the option of not specifying any callback function (Rsphdl) when the command is executed. Specify NULL in place of the callback function. The adl_atCmdCreate() API should then be invoked as:
adl_atCmdCreate("command",TRUE,NULL,NULL);

8.2. adl_atCmdSend () API

This API functionality is same as adl_atCmdCreate without the rspflag argument and instead it sends the command to the required command port stack, in order to be executed as soon 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 is returned on success.

ERROR is returned in case of an error.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

8.3. adl_atCmdSendExt () API

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

Prototype:

s8 adl_atCmdSendExt (ascii * atstr,adl_atPort_e port,u16 NI,ascii * Contxt,adl_atRspHandler_t rsphdl,...)

Parameters:

atstr: The string (name) of the command you 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 provided by the command handler when re sending the command already subscribed to solve any loop effect. 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

Contxt: Context made to hold information gathered at the time the command wassent.

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 is returned on success.

ERROR is returned in case of an error.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

8.4. adl_atCmdSendText () API

This API is used to send intermediate text for a command which requires intermediate input text (For e.g. AT+CMGS).

Prototype:

s8 adl_atCmdSendText (adl_port_e Port, ascii* Text)

Parameters:

Port: This parameter determines the port on which the text string should be sent. The port can be a physical or virtual port. The values this parameter can take are defined in the adl_port_e enum.

Text: This parameter determines the text string to send to the specified port as intermediate text. If the text doesn't end with CTRL+Z character, it is added automatically by this API.

Returned values:

OK is returned if the response is sent successfully.

ERROR value is returned in case of an error.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTES:
dl_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 subscriptions, 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.

For any multiply subscribed command sent by an external application on one of the embedded modules 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 ...).
If any Inner AT Command (as decribed in section 2.5 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 2.5 of ADL UGD it may affect ADL correct behaviour.
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

8.5. Sample Code

```
// 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);
}
```

8.6. Sample Code

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

bool CMGS_Handler (adl_atResponse_t *paras)
{
  /* If ”>” received for the CMGS, send the intermediate text */
  if (!wm_strncmp (paras->StrData, ”\r\n>”, 3))
  {
    /* Send the intermediate text */
    adl_atCmdSendText(ADL_PORT_UART1, “This is a test message” );
  }
  if ( paras->RspId == ADL_STR_OK )
  {
    /* Send a response to the user */
    adl_atSendResponsePort (ADL_AT_RSP, ADL_PORT_UART1, ”\r\nSMS sent
                                  succesfully”);
  }
  return TRUE;
}

bool wind_4_handler(adl_atUnsolicited_t * paras)
{
  TRACE((1,"Inside wind 4 handler"));
  adl_atCmdCreate(“AT+CMGS=123456780”, ADL_AT_PORT_TYPE(ADL_PORT_UART1, TRUE),
                CMGS_Handler , “*”, NULL)
  return (0);
}
```

```
void adl_main ( adl_InitType_e InitType )
{
  TRACE((1,"Inside adl_main"));
  adl_atUnSoSubscribe("+WIND: 4",wind_4_handler);
}
```

8.7. Creating Custom AT Commands

Open AT OS allows you to create and subscribe to your own custom AT commands. The following sections describe the Open AT APIs that you can use to create new AT commands
The following API can be used to create a custom AT command:

Prototype:

s16 adl_atCmdSubscribe (ascii *Cmdstr, adl_atCmdHandler_t Cmdhdl, u16 Options)
or
s16 adl_atCmdSubscribeExt (ascii *Cmdstr, adl_atCmdHandler_t Cmdhdl, u16 Options, void *Context, adl_atCmdSubscriptionPort_e CmdPort)

This API allows you to subscribe (create) to a specific command along with an associated callback handler, so that whenever this command is sent by the external application, the callback handler is executed. It should be noted that the subscribed command can be a standard AT command or a new user defined AT command. In this way, you can create new AT commands. However, these custom commands can be executed only when the Open AT Application is running.

Parameters:

Cmdstr: The string or the command to which you want to subscribe.

Cmdhdl: This parameter is the callback function which must be executed whenever the subscribed command is sent by the external application. This function has the following prototype:
typedef void (*adl_atCmdHandler_t) (adl_atCmdPreParser_t *)
The argument of this callback function is of the adl_atCmdPreParser_t type and holds the command and its parameters to which you have subscribed. It has the following definition:

```
typedef struct
{
u16 Type;                 //Type of command
u8   NbPara;                    //No of valid parameters (different
                    //from “”)
adl_atPort_e Port;      // Port
wm_lst_t ParaList;            // Parameter list for the command
u16 StrLength;            //Length of command
u16 NI;                //Notificaiton identifier
void* Context;          //Context
ascii StrData[1];      //Pointer to the command string
} adl_atCmdPreParser_t;
```

Whenever the subscribed command is sent by the external application the callback function is called and the elements of the adl_atCmdPreParser_t structure are filled with arguments provided to the command.

Parameters:

Type: This member indicates the type of command that is sent. The commands are also classified on the basis of their operation and whether or not they provide any input (using arguments). The following types of commands are defined.

Command Type	Value	Meaning Assigned
ADL_CMD_TYPE_PARA	0x0100	These types of commands allow arguments to be passed. For example, the AT+CMD=x,y command. However, the command must have the correct format for the callback to be executed.
ADL_CMD_TYPE_TEST	0x0200	These types of commands allow you to get the values which can be set for its parameter. For example, AT+CMD=?.
ADL_CMD_TYPE_READ	0x0400	This type of commands allows you to test the value already set for the parameter. For example, AT+CMD?.
ADL_CMD_TYPE_ACT	0x0800	This is the simple type of commands which can be subscribed. For example, "AT+CMD".
ADL_CMD_TYPE_ROOT	0x1000	This type allows you to subscribe to all the commands starting with a particular string. The callback handler will receive the whole AT string.
ADL_CMD_TYPE_ROOT_EXT	0x2000	This parameter is same as ADL_CMD_TYPE_ROOT in this version.

NbPara: This member contains the number of valid arguments (those parameters which are different from "") of the command. This is valid only in case the command is of the ADL_CMD_TYPE_PARA type as this is the only command type which can have arguments.

Port: This member contains the port information on which the command is sent by the external application.

ParaList: This member contains the parameters which are passed to the command (in case the command is of the ADL_CMD_TYPE_PARA) type. To retrieve the parameters, use the ADL_GET_PARAM (_P_,_i_) macro. The first parameter (_P_) of this macro is a pointer to the adl_atCmdPreParser_t structure and the second parameter (_i_) is the requested parameter index (starting from 0). This macro (ADL_GET_PARAM ()) returns a pointer to the ASCII string which contain the arguments passed in the command.

StrLength: This member contains the length of the command to which you have subscribed. Whenever, this command is sent by the external application, the StrLength parameter will contain the complete length of the string entered (which constitutes the command).

NI: This parameter is to hold the Notification Identifier provided by the command handler when re sending the command already subscribed to solve any loop effect.

Context: This parameter contains the context information which is provided during the subscription.

StrData: This member is the pointer to the string of the command which is entered through the external application.

Options:
This parameter is used to provide the command type (ADL_CMD_TYPE_PARA, ADL_CMD_TYPE_READ, ADL_CMD_TYPE_ACT, ADL_CMD_TYPE_ROOT, and ADL_CMD_TYPE_TEST) information to the API. If the command type is ADL_CMD_TYPE_PARA (which accepts parameter), then the Options parameter also provide the maximum and minimum parameters which can be used with the command. This is done with the logical OR of the following information:
The minimum number of arguments 'a' is stored in the least significant nibble (4 bits) as in 0x000a.

The maximum number of arguments 'b' which can be provided to the command should be stored in the second least significant nibble as in 0x00b0.

For example, if a command expects a maximum of 4 arguments and a minimum of 1 argument then this can be specified in the Options parameter by giving ADL_CMD_TYPE_PARA|0x0041 as its value.

Context: This parameter holds context information that will be gathered during the command subscription.

Cmdport: This parameter defines the port on which the command is subscribed. This is defined by ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT and ADL_CMD_SUBSCRIPTION_ALL_PORTS. Note that in this version only ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT valid value for this option.

Returned Values:

OK is returned on successful execution of the API.

ERROR is returned if an error has occurred.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTES
Only those commands which accept parameters (ADL_CMD_TYPE_PARA) should specify the maximum and minimum number of arguments.
You can also subscribe to standard AT commands and provide new functionality to these commands (in the callback function). In this way, your Open AT Application can become extremely flexible.
To prevent the callback function from being called every time the command is sent, unsubscribe from the command.
If you subscribe to a command with a callback handler (for example func1()) and then subscribe the same command with a callback handler (for example func2()), then whenever this command is issued, both the handlers(func1() and func2()) will be invoked depending upon the subscription order
ADL internally can subscribe to few AT commands and these commands shouldn't be subscribed by Open AT

Application. Subscribing to these commands may result in undefined behaviour. These commands include:
- AT+CPIN
- ATD, ATA, ATH
- AT+CGACT, AT+CGANS, AT+CGDATA
- AT+WDWL, AT+WOPEN

ATO, ATQ, ATV, AT+WIND, AT+CMEE, AT&W, ATZ, AT&F, AT+CREG, AT+CGREG, AT+CRC, AT+CGEREP

8.8. Unsubscribing AT Commands

Open AT OS provides the following API to unsubscribe from custom AT commands that you no longer want to use.

Prototype:

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

After calling this API, the callback function associated with the subscribed command will not be executed again.

Parameters:

Cmdstr: This parameter contains the string (name) of the command from which you want to unsubscribe. This might contain a standard AT command, or a user-defined AT command.

Cmdhdl: This parameter is the callback function associated with the previously subscribed command.

Returned Values:

OK is returned if the command was found and successfully unsubscribed.

ERROR is returned if the command was not found.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTES:
Only the commands subscribed using adl_atCmdSubscribe () API can be unsubscribed using adl_atCmdUnSubscribe ().
After unsubscribing the command, the callback function will not be called when the command is issued again from the external application

8.9. Setting Quiet Mode

Open AT OS provides the following API to set Quite mode. This mode has the behaviour similar to ATQ command. In this mode, the terminal response will not be sent to external application.

Prototype:

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

Parameters:

IsQuiet: This parameter sets the quiet mode. The quite mode will be enabled if this parameter is set to TRUE. If it is set to FALSE then the quite mode will be disabled.

Returned Values:

None

8.10. Sample Code

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

void command_handler(adl_atCmdPreParser_t *paras)
{
 ascii buffer[35];
 switch (paras->Type)
 {
   case ADL_CMD_TYPE_READ:
     adl_atSendResponse(ADL_AT_RSP,"\r\nIssued AT+TEST?\r\n");
     break;
   case ADL_CMD_TYPE_TEST:
     adl_atSendResponse(ADL_AT_RSP,"\r\nIssued AT+TEST=?\r\n");
     break;
   case ADL_CMD_TYPE_ACT:
     adl_atSendResponse(ADL_AT_RSP,"\r\nIssued AT+TEST\r\n");
     break;
   case ADL_CMD_TYPE_PARA:
     wm_strcpy(buffer,"\r\nIssued AT+TEST = ");
     wm_strcat(buffer,ADL_GET_PARAM(paras,0));
  wm_strcat(buffer,"\r\n");
  adl_atSendResponse(ADL_AT_RSP,buffer);
  break;
}
adl_atSendResponse(ADL_AT_RSP,"\r\nOK\r\n");
}

void adl_main ( adl_InitType_e InitType )
{
  TRACE((1,"Inside adl_main"));
  adl_atCmdSubscribe("AT+TEST", (adl_atCmdHandler_t)command_handler,
ADL_CMD_TYPE_TEST|ADL_CMD_TYPE_READ|
                                                        ADL_CMD_TYPE_ACT|ADL_CMD_TYPE_PARA|0x0011);
}
```

Summary

The following points have been covered in this chapter

- **Different types of responses: Terminal Responses, Intermediate Responses and Unsolicited Responses.**
- **To subscribe and unsubscribe unsolicited responses use the adl_atUnSoSubscribe () and adl_atUnSoUnSubscribe () APIs respectively.**
- **To send responses from the Open AT Application use the adl_atSendResponse (), adl_atSendStdResponse () and adl_atSendStdResponseExt () APIs.**
- **To send standard AT commands use the adl_atCmdCreate ()/ adl_CmdSend ()/ adl_CmdSendExt ()/ adl_CmdSendText () API from within the Open AT Application.**
- **To subscribe and unsubscribe AT commands use the adl_atCmdSubscribe ()/ adl_atCmdSubscribeExt () and adl_atCmdUnSubscribe () APIs**

CHAPTER 8

AT Services

1. Objective

This chapter introduces you to the AT strings service and the functions that you can use to process the strings.

2. AT Standard String Set

The AT standard string set is a set of responses received in response to commands issued to the embedded module. The adl_strID_e Enum data type lists all the AT response strings IDs and is enumerated as shown below:

```
typedef enum
{
ADL_STR_NO_STRING,       // The string is unknown
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,     // +CMSERROR:
ADL_STR_CPIN,           // +CPIN:
ADL_STR_LAST_TERMINAL, //ALL TERMINAL RESPONSE ARE BEFORE THIS STRING
ADL_STR_RING= ADL_STR_LAST_TERMINAL, //RING
ADL_STR_WIND,  //+WIND: INDICATION
ADL_STR_CRING,               //+CRING: INDICATION
ADL_STR_CPINC,               //+CPINC: INDICATION
ADL_STR_WSTR,  //+WSTR: INDICATION
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. APIs

3.1. Getting a String Response ID

The adl_strGetID () function can be used to get the string ID corresponding to a given response string. The ID obtained can be used to make the programming logic simpler. For instance, while using a switch-case statement, it is simpler to use an ID instead of an actual string.

Prototype:

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

Parameters:

rsp: String to be parsed to get the ID.

Returned Values:

This function returns the ID of string parsed.

ADL_STR_NO_STRING - if the string is unknown.

3.2. Getting an ID along with an Optional Response Argument

This function is used to get the ID of the provided response string along with an optional argument of the response string. The function is defined below:

Prototype:

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

Parameters:

rsp: String to be parsed to get the ID.

arg: This parameter takes the value of parsed argument of the string. For example, the ADL_STR_CME_ERROR string when parsed, will return the string ID along with the CMEE error code. The arg parameter will return the CMEE error code value. For example, if the response is +CME ERROR: 10, the arg parameter is 10.

argtype: Type of the parsed argument:

If the argument type is ADL_STR_TYPE_ASCII, then the argument is an Ascii * string;
If the argument type is ADL_STR_TYPE_U32, then the argument is a U32 *Integer

Returned Values:

This function returns the ID of string to be parsed.

ADL_STR_NO_STRING: if the string is unknown.

3.3. Checking for the Last Response String

The following function can be used to check whether a particular response string is the last string that was received for a sent command. This function can be used to check for any of the terminal responses received as well, and the return value can be used as a condition to implement a particular logic.

Prototype:

```
bool adl_strIsTerminalResponse( adl_str_ID_e RspID )
```

This function checks whether a particular response ID belongs to the last response.

Parameters:

RspID: the response ID that must be checked.

Returned Values:

TRUE: if the provided response ID belongs to the last response.
FALSE: if the provided response ID does not belong to the last response.

3.4. Getting a Response String for an ID

Given an ID, the adl_strGetResponse () function is used to get the AT response string corresponding to that ID. The function is described below.

Prototype:

```
ascii * adl_strGetResponse ( adl_str_ID_e RspID )
```

Parameters:

RspID: This parameter takes the ID corresponding for which the response string must be found.

Returned Values:

This function returns the response string.

NULL: if the ID does not exist.

3.5. Getting a Response String and an Argument for an ID

This function is used to retrieve the response string from a given ID, in cases where the string responses have an optional argument.

Prototype:

```
ascii * adl_strGetResponseExt ( adl_str_ID_e RspID, u32  arg )
```

Parameters:

RspID: This parameter takes the ID corresponding to which the response string is to be found.

arg: this parameter gives the response ID to be copied in the response string. According to response ID, this argument should be an integer value, or an ascii *string.

Returned Values:

This function returns the response string

NULL – if the ID does not exist.

4. Sample Code

```
#include "adl_global.h"
#if __OAT_API_VERSION__ >= 400
const u16 wm_apmCustomStackSize = 4096;
#else
u32 wm_apmCustomStack[1024];
const u16 wm_apmCustomStackSize = sizeof(wm_apmCustomStack);
#endif
/* Local variables */
ascii *arg;
u8 *argType;
bool Termresp=FALSE;
ascii * rsp="+WIND: 4";
ascii * rspArgStr="+CME ERROR: 10";
adl_strID_e Get_ID;
adl_strID_e StrIDArg;
u32 Get_String=2;
ascii * ResponseStr;
ascii * ResponseStrArg;
u32 argresp;
adl_tmr_t * timerHandle;

 /* Function  : adl_main */
```

```
void adl_main ( adl_InitType_e InitType )
{
            arg=(ascii *)adl_memGet(4);
            argType=(ascii *)adl_memGet(20);
            wm_memset(arg,0,4);
            wm_memset(argType,0,20);
            TRACE (( 1, "Embedded Application: Main" ));
            /* Function to get ID  from a given response string */
            Get_ID= (adl_strID_e)adl_strGetID (rsp );
            TRACE( ( 1, "String ID for 'OK'= %d",Get_ID ) );

/* Function to get response string ID and argument from a given response string */
StrIDArg = (adl_strID_e) adl_strGetIDExt (rspArgStr,arg , argType );
            TRACE( ( 1, "String ID for '+CME ERROR: 10'= %d",StrIDArg ) );
            TRACE( ( 1, "argument= %d",*arg) );

            /* Function to get ID  from a given response string */
Termresp = (adl_strID_e)adl_strIsTerminalResponse ( ADL_STR_CREG  );//s
            TRACE( ( 1, "terminal Response Bool value1= %d",Termresp) );//s
Termresp = (adl_strID_e)adl_strIsTerminalResponse ( ADL_STR_OK  );//s
            TRACE( ( 1, "terminal Response Bool value2= %d",Termresp) );//s

            /* Function to get response string from a given response string ID */
ResponseStr = adl_strGetResponse (Get_String);
adl_atSendResponse(ADL_AT_UNS,"\r\nResponse string for a given ID\r\n");
            adl_atSendResponse(ADL_AT_UNS,ResponseStr);

/* Function to get response string along with argument from a given response string ID*/

            ResponseStrArg = adl_strGetResponseExt ( 7,10 );
adl_atSendResponse(ADL_AT_UNS,"\r\nResponse string along with arg for a given ID\r\n");
adl_atSendResponse(ADL_AT_UNS,ResponseStrArg);
}
```

Summary

The following points have been covered in this chapter

- **Open AT OS provides APIs to process AT standard string responses. The need for AT strings service arises when certain functionality must be implemented based on the AT responses received from various AT command.**
- **To get an ID from a response string use adl_strGetID () function.**
- **To get an ID from a response string containing an optional argument use adl_strGetIDExt ()function**
- **To check whether or not the given string belongs to the last response use the adl_strIsTerminalResponse()function.**
- **To get a response string for a given response ID use the adl_strGetResponse()function.**
- **To get a response string which has an optional argument, use the adl_strGetResponseExt function**

CHAPTER 9

Timer Service

1. Objective

The objective of this chapter is to introduce you to timers and their usage. Timers are extensively used in Open AT Applications to introduce a time delay, execute a process or poll for events. This can prove to be handy in situations where a time delay is necessary between subsequent instructions or network operations as depicted in the examples that follow in this chapter.

This chapter addresses the following questions:

- What are timers?
- How are timers classified?
- How are timers used?

2. Timers

Timers are indispensable components of the Open AT OS. Timers provide the following functionality:

- Run a function/procedure after a specified period of time.
- Automatically execute a task at periodic intervals.
- Hence, timers can be used when:

A process needs to be executed repeatedly after a specific interval. For example, polling of an input pin, or sending sensor data to the server every hour.

A delay is needed in the execution of statements. For instance, while dealing with network-related activities, a time delay must be introduced so that the outcome of the first activity is available before the second activity is initiated. In these cases, timers can provide the necessary delay.

3. Types of Timers

Two types of timers are available in Open AT OS: Cyclic and Non-Cyclic Timers.

3.1. Cyclic Timers

These timers can run a particular process repeatedly after a specified time interval. These have the following advantages:

- Cyclic timers help to automate the repetitive tasks.
- They help in reducing the code size by automatically calling the timer handler functions; you do not need to duplicate the corresponding lines of code.

3.2. Non-Cyclic Timers

These timers are used to provide a one-time delay in executing a particular process, and are not scheduled to run again.
For instance, an Open AT Application can be put on hold for sometime during its initialization to give the network registration operations time to be completed before the Open AT Application starts other functions.

4. Using Timers in Open AT Applications

Open AT OS provides APIs which simplify the use and manipulation of timers. These APIs are declared in the adl_TimerHandler.h header file. This file must be included in the Open AT Application which implements timers.

4.1. Subscribing and Starting a Timer

The following API is provided to subscribe to a timer (Cyclic/Non-Cyclic) and associate a process with it (callback function):

Prototype:

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

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

This API starts a timer with an associated callback function. The callback function will be executed when the timer interval expires. The statements which constitute the process to be scheduled should be placed in the callback function so that they will be automatically executed after the time interval expires.

Prototype:

bCyclic: This Boolean parameter determines whether a cyclic or a non-cyclic timer must be started. If this parameter is set to TRUE, a cyclic timer will be started. If this parameter is set to FALSE, then a Non-Cyclic timer will be started.

CyclicOpt: This parameter determines whether a cyclic or a non-cylcic timer must be started. This is defined by the following structure:

```
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;
```

Parameters:

ADL_TMR_CYCLIC_OPT_NONE: The timer will be started in non-cyclic mode if this parameter is used for setting the timer mode. The timer will be automatically unsubscribed as soon as the timer expiration event is notified to the application.

ADL_TMR_CYCLIC_OPT_ON_EXPIRATION: The timer will be started in cyclic mode if this parameter is used for setting the timer mode. The timer will be re-programmed before the timer expiration event is sent to the application. Hence, there is no minimum time guaranteed between two timer events, since if the application is pre-empted for some time, timer events will continue to be generated even if the application is not notified.

ADL_TMR_CYCLIC_OPT_ON_RECEIVE: The timer will be started in cyclic mode if this parameter is used for setting the timer mode. The timer will be re-programmed after the timer expiration event is sent to the application. Hence, the duration between two events is guaranteed to be at least equal to the timer period.

ADL_TMR_CYCLIC_OPT_LAST: This parameter is reserved for internal use.

TimerValue: This parameter indicates the period after which the timer expires. This parameter is dependent on the TimerType parameter.

TimerType: This parameter represents the unit of TimerValue parameter. In other words, this parameter is the unit of the timeout interval. The possible values which this parameter can take are:

ADL_TMR_TYPE_100MS: This value indicates that the TimerValue is in 100 milliseconds steps. The least timeout value which can be set using the timer is 100 milliseconds. The TimerValue parameter will indicate the steps of 100 milliseconds after which the timer expires. For example, if the TimerValue has a value of 2, then the timer will expire after 200 milliseconds (2 * 100 milliseconds).

ADL_TMR_TYPE_TICK: This value indicates that the TimerValue is in 18.5 millisecond steps. Hence, the least value which can be set using the timer will become 18.5 milliseconds. The TimerValue parameter indicates the steps of 18.5 milliseconds after which the timeout occurs. For example, if TimerValue parameter has a value of 2, then the timer will expire after 2 * 18.5 = 37 milliseconds.

Timerhdl: This parameter is the callback function which is called after timeout. For cyclic timers, this function is called repeatedly after each timeout. For non-cyclic timers, this function is called only once after the timeout occurs. This function has the following prototype:

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

The argument received by this function will contain the timer ID and the Context. There can be a maximum of 32 timers running simultaneously. The timer ID is an integer parameter that will differentiate a timer from other timers. The maximum value of the Timer ID that can be received in the timer callback function is 31; the minimum value is 0. The timer ID values are assigned in a circular fashion. That is, if a timer with a timer ID of 1 is subscribed, the next subscription will yield a timer ID 2, irrespective of whether or not the previous timer was subscribed. After the whole range (0-31) is used, the timer IDs will be reused.

The argument context is a pointer on the application context provided to adl_tmrSubscribeExt (). This will be set to NULL if the timer is subscribed using adl_tmrSubscribe ().

Context: This is a 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.

Returned Values:

If the API is successful, it returns the pointer to the timer which is started. This pointer is used to unsubscribe the timer.

ERROR is returned if there is a problem in starting the timer.

NULL is returned if the timer value is zero or if there is no more timer resource available.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTE :
There can be a maximum of 32 timers running at a time. If a request is made for a new timer after subscribing for all 32 timers, the adl_tmrSubscribe/adl_tmrSubscribeExt API returns NULL.
A single callback function can be associated with several timers.
The minimum delay which can be provided using a timer is 18.5 milliseconds.

4.2. Unsubscribing a Timer

To unsubscribe a timer, Open AT OS provides the following API:

Prototype:

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

This API stops and unsubscribes the timer and its handler. The call to this API is only meaningful for a cyclic timer or a timer that hasn't expired yet. After execution of this API, the callback function of the timer will not be called again.

Parameters:

tim: This parameter is the pointer to the timer from which you want to unsubscribe. This is received as a return value of adl_tmrSubscribe API.

Timerhdl: This parameter is the callback function associated with the timer.

TimerType: This parameter represents the unit of timeout interval. This parameter can have the following values:

ADL_TMR_TYPE_100MS: The TimerValue is in 100 milliseconds steps.
ADL_TMR_TYPE_TICK: The TimerValue is in 18.5 milliseconds steps.

Returned Values:

If the API is successful, it returns the remaining time of the timer before the timeout occurs. (The unit will be according to the TimerValue parameter).

ADL_RET_ERR_BAD_HDL is returned if the provided timer handle is unknown or not in accordance with the handler.

ADL_RET_ERR_BAD_STATE is returned if the handler (callback function) has already expired.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTE :
The call to adl_tmrUnSubscribe API is only meaningful for a cyclic timer or a timer that hasn't expired yet. A non-cyclic timer automatically unsubscribes itself when the timer times out.

5. Sample Code

5.1. Subscribing to a non-cyclic timer

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

adl_tmr_t *timer_ptr;
u32 timeout_period = 5;

void Timer_Handler (u8 Id, void *context)
{
  TRACE ((1,"Inside timer handler. Timer id is %d", Id));
  adl_atSendResponse (ADL_AT_UNS,"\r\nTimer expired");
}

void adl_main (adl_InitType_e adlInitType)
{
  // A Non Cyclic timer is subscribed.
  //There is no need to unsubscribe the timer.
  timer_ptr = (adl_tmr_t*) adl_tmrSubscribe (FALSE, timeout_period,
            ADL_TMR_TYPE_100MS, (adl_tmrHandler_t) Timer_Handler);
}
```

5.2. Subscribing to a cyclic timer

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

adl_tmr_t *timer_ptr;
u16 timeout_period = 5;
static counter=0;

void Timer_Handler (u8 Id, void* Context)
{
  TRACE ((1,"Inside timer handler. Timer id is %d", Id));
  counter++;
  adl_atSendResponse (ADL_AT_UNS,"\r\nCyclic Timer expired");
  if (counter==50)
  {
    adl_tmrUnSubscribe (timer_ptr, (adl_tmrHandler_t)
    Timer_Handler, ADL_TMR_TYPE_100MS);
  }
}
void adl_main (adl_InitType_e adlInitType)
{
  // A cyclic timer is subscribed.
  timer_ptr = (adl_tmr_t*) adl_tmrSubscribe (TRUE, timeout_period,
            ADL_TMR_TYPE_100MS, (adl_tmrHandler_t) Timer_Handler);
}
```

Summary

The following points have been covered in this chapter

- **Timers are tools which allow a function to be called at regular interval and induce a time delay in the execution of statements.**
- **There are two types of timers:**
    - **Cyclic Timers: These timers can be used when periodic execution of task is required.**
    - **Non-Cyclic Timers: These timers are used to provide a one time delay in executing a particular task.**
- **To subscribe to a timer use adl_tmrSubscribe ()/ adl_tmrSubscribeExt() API.**
- **To unsubscribe a timer use adl_tmrUnSubscribe () API**

CHAPTER 10

Error Management Service

1. Objective

This chapter describes how to manage error messages in the embedded module using the Open AT OS.

2. Introduction

Open AT OS uses a special area in flash to store the error description and call stack when an exception occurs. This area is called backtraces and the data inside this area can be retrieved using Error Management service APIs.
The error management service also allows user to catch the error generated through the Open AT Application. This service also allows raising the custom errors from the Open AT Application. When this error occurs, the embedded module may reset.

3. Error Management APIs

This section describes the APIs that are available in the embedded module for error management. Include the adl_error.h header file to use the error management APIs. The APIs discussed below are:

adl_errSubscribe
adl_errUnsubscribe
adl_errHalt
adl_errEraseAllBacktraces
adl_errStartBacktraceAnalysis
adl_errGetAnalysisState
adl_ errRetrieveNextBacktrace

3.1. Subscribing to the Error Management Service

The following API subscribes to the error management service:

Prototype:

s8 adl_errSubscribe(adl_errHdlr_f ErrorHandler)

Parameters:

ErrorHandler: This Error handler is invoked when a error occurs; this function is declared thus:

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

An error is associated with the ID (ErrorID) and the corresponding text string (ErrorStr), sent as parameters to the adl_errHalt API.

Returned Values:

FALSE: if the error is processed and filtered.

TRUE: will cause the embedded module to execute a fatal error reset and give a backtrace.

Returned Values:

OK is returned on successful subscription to the error management service.

ADL_RET_ERR_PARAM is returned if a parameter has an incorrect value.

ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the service is already subscribed.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

NOTE :
ErrorID below 0x0100 are for internal purpose. So you must only use ErrorIDs above 0x0100.
You cannot subscribe to the error management service more than once at a time. You have to unsubscribe before subscribing again

3.2. Unsubscribing from the Error Management Service

The following API should be used to unsubscribe from the error management service:

Prototype:

```
s8 adl_errUnsubscribe( adl_errHdlr_f ErrorHandler )
```

Parameters:

ErrorHandler: The same handler passed to the adl_errSubscribe API.

Returned Values:

OK is returned on successfully unsubscribing from the error management service.

ADL_RET_ERR_PARAM is returned if a parameter has an incorrect value.

ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handler is unknown.

ADL_RET_ERR_NOT_SUBSCRIBED is returned if the service is not subscribed.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

3.3. Handling an Error

The following function causes an error that is defined by its ID and string:

Prototype:
void adl_errHalt(u16 ErrorID, const ascii * ErrorStr)

If an error handler is defined, it is invoked when this API is called, otherwise, the API will result in resetting the embedded module.

Parameters:
ErrorID: ID of the error.

ErrorStr: Error string to be passed to the error handler.

Returned Values:
None

3.4. Start Backtrace analysis

This API is used to start the backtrace analysis process. This process has to be started before Open AT Application can read the backtraces. When the embedded module is rebooted due to exception, this process can be started to understand the cause of the exception.

Prototype:
s8 adl_errStartBacktraceAnalysis (void)

Parameters:
None

Returned Values:
Positive or null handle is returned on success.

ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if analysis is already running.

ERROR is returned if backtrace analysis has failed to start.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

3.5. Retrieve next Backtrace

This API is used to retrieve the next backtrace stored in the non-volatile memory. The analysis process has to be started before using this API.

Prototype:
s32 adl_errRetrieveNextBacktrace (u8 Handle, u8 * BacktraceBuffer, u16 Size)

Parameters:

Handle: Handle returned by adl_errStartBacktraceAnalysis () API.

BacktraceBuffer: Buffer where next retrieved backtrace will be copied. If mentioned NULL, this API will return the required size for the next call to this API.

Size: Size of the BacktraceBuffer

Returned Values:

OK is returned on success.

Size is the required backtrace buffer size if BacktraceBuffer is NULL.

ADL_RET_ERR_PARAM is returned if size is not large enough.

ADL_RET_ERR_NOT_SUBSCRIBED is returned if backtrace analysis has not started.

ADL_RET_ERR_UNKNOWN_HDL is returned if incorrect handle is mentioned.

ADL_RET_ERR_DONE is returned if there are no more backtraces to read.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

NOTE :
Back trace retrieved using adl_errRetrieveNextBacktrace () should be sent using FCM flow to the external application. GPRS flow can also be used to send the traces to the remote system.
These backtraces are ciphered, so an Open AT Application can't interpret them.

3.6. Erase all backtraces

This API is used to erase all the stored backtraces in the non-volatile memory. If the backtrace memory is full, the backtraces are not over-written, hence it is very important to erase the backtraces using this API.

Prototype:

s32 adl_errEraseAllBacktraces (void)

Parameters:

None

Returned Values:

OK is returned on success.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

3.7. Get the current analysis state

This API is used to get the current backtrace analysis state.

Prototype:

adl_errAnalysisState_e adl_errGetAnalysisState (void)

Parameters:

None

Returned Values:

Return the current analysis state. The values it can take are defined in the adl_errAnalysisState_e enum. This enum is defined below:

```
typedef enum
{
                ADL_ERR_ANALYSIS_STATE_IDLE,     // No running analysis
                ADL_ERR_ANALYSIS_STATE_RUNNING  // A backtrace analysis is
                                                  // running
}adl_errAnalysisState_e
```

4. Sample Code

```
/* sample code implementing the error management functionality */
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

#define MY_CSQ_ERRNUM 512
adl_tmr_t * tmr_hdl;

//ascii * strGetParameterString ( ascii * dst, const ascii * src, u8 Position );

/* Called whenever an error occurs*/
bool ErrHandler ( u16 ErrorID, ascii* ErrorStr)
{
  TRACE((1,"Error handler: Error ID = %d", ErrorID));
  TRACE((1,ErrorStr));
  if( ErrorID == ADL_ERR_LEVEL_APP )
  {
    return FALSE;
  }
  return TRUE;
}

/* Timer handler */
void TimerHdlr( u8 ID, void *context)
{
  static u8 count = 0;
  count++;
  if( count == 10 )
  {
   adl_tmrUnSubscribe(tmr_hdl,TimerHdlr,ADL_TMR_TYPE_100MS);
   adl_errHalt(ADL_ERR_LEVEL_APP,"Error From Appl");
  }
}

/* Main function */
void adl_main ( adl_InitType_e InitType )
{
  s8 ret;
  TRACE((1,"Main function"));
  ret = adl_errSubscribe(ErrHandler);
  TRACE((1,"Return value of adl_errSubscribe=%d",ret));
  tmr_hdl = adl_tmrSubscribe(TRUE, 30, ADL_TMR_TYPE_100MS, TimerHdlr);
}
```

Summary

The following points have been covered in this chapter

- **The APIs for error management are defined in the adl_error.h header file.**
- **The adl_errSubscribe API should be used to subscribe to the error management service.**
- **The adl_errUnsubscribe API should be used to unsubscribe from the error management service.**
- **The adl_errHalt API should be used to control the embedded module behaviour when an error occurs. For fatal error, a reset could be performed and for a non fatal error a trace could be performed.**
- **The adl_errStartBacktraceAnalysis API should be used to start the backtrace analysis process.**
- **The adl_errRetrieveNextBacktrace API used to retrieve the next backtrace stored in the non-volatile memory.**
- **The adl_errEraseAllBacktraces used API to erase all the stored backtraces in the non-volatile memory.**
- **The adl_errGetAnalysisState API used to get the current backtrace analysis state.**

CHAPTER 11

Memory Management

1. Objective

The objective of this chapter is to introduce the different sections of RAM available in the embedded module and the usage of each in detail.

2. Sections in RAM

RAM available for the embedded module has two sections. They are:

- Open AT OS RAM
- Firmware RAM

Firmware RAM is managed by Real Time Kernel and Open AT RAM is the memory space available for Open AT Application and is discussed in more detail in the following sections

3. Open AT OS RAM

The objective of this section is to introduce the usage of RAM memory in Open AT Application. Memory is always an important and scarce resource in any embedded device. In Open AT Application memory could be allocated dynamically (at runtime). It is essential that you release the memory once it has been used.
This chapter provides answers to the following queries:

- How can you allocate memory?
- How can you release memory?
- How to get the current type of the memory?
- How to retrieve the current RAM information?

4. Memory Management in Open AT OS

The total RAM area is divided in three parts

Stack: Customer defines the size of the area in application.

Data (Global Variables): This area depends on number of global and static variables used in the application.

Heap: Customer can use memory management APIs to allocate/release memory

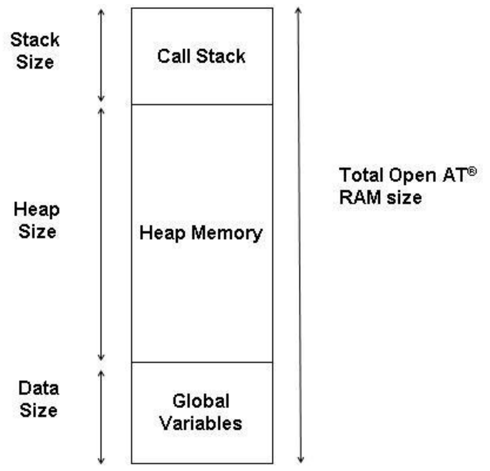


Figure 51: Diagrammatic representation of partitioning of RAM

The map file (file name: m) can be used to observe the size of different areas in RAM and Flash. This file is generated during Open AT Application binary generation and is available in the "out" folder. Please find below the extract from the map file generated for Hello World sample:

```
==============================================================
   Code          RO Data          RW Data          ZI Data          Debug
  39036      1678          420                    586      82152   Grand Totals
==============================================================
  Total RO  Size(Code + RO Data)                            40714 (  39.76kB)
  Total RW  Size(RW Data + ZI Data)                         1006 (   0.98kB)
  Total ROM Size(Code + RO Data + RW Data)        41134 (  40.17kB)
==============================================================
```

Here RAM allocated for global variables is 0.98 kB. This includes RW data of 420 bytes for uninitialized global variables and ZI data of 586 bytes for zero initialized global variables.
The RO (Read Only) data is added to the Open AT Application binary which is later flashed to embedded module. The RO data includes constant variables used in the Open AT Application.

5. Allocating Memory

Memory APIs are declared in the adl_memory.h header file. Hence, this file must be included in the Open AT Application which requires memory to be allocated dynamically.
For allocating memory, Open AT OS finds the best block in heap memory and allocates to the customer. If the block is unavailable, it returns an error.

The user should take care of fragmentation issues that could occur due to frequent allocation/release of memory. For efficient use of heap memory, user can allocate heap memory of constant size. For e.g. user can define the memory size as 128, 512, 1024 and so on. When the application needs memory of 100 bytes, it can allocate size of 128 bytes.

The following API can be used to dynamically allocate memory of the requested size in Open AT Application's RAM memory:

Prototype:

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

Parameters:

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

Returned Values:

A pointer to the allocated memory.

ADL_ERR_MEM_GET error in case the memory allocation fails, which is handled by the error service. If the error handler filters and refuse this error, then the function returns NULL.

NOTE :
adl_memGet API throws an exception (ADL_ERR_MEM_GET) if there is a failure in memory allocation.

6. Releasing Memory

6.1. The adl_memRelease macro

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

```
#define adl_memRelease (_p_)   adl_memRelease ((void **) &_p_)
```

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.

6.2. 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);
```

This API releases the memory allocated to the supplied pointer and sets the pointer to NULL.

Parameters:

ptr: Pointer to the allocated memory buffer address.

Returned Values:

Please refer to the adl_memRelease macro definition

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.*

6.3. The ADL_MEM_UNINIT macro

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.

Prototype:

```
#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 the variable declaration, there is no semicolon
Global variable cannot be initialized with a value when it is declared

Warning:
This macro 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
 ...
}
```

6.4. Get the RAM information

This API returns the information about the Open AT OS RAM area sizes such as Heap stack and global. This information can be retrieved dynamically; however it doesn’t return the remaining or allocated size of heap memory. This APIs returns the size of total heap memory along with size of other RAM areas.

Prototype:

```
s32 adl_memGetInfo( adl_memInfo_t * Info )
```

Parameters:

Info: Structure is updated by the API. The structure is defined below:

```
typedef struct
{
  u32 TotalSize  // Total RAM size for Open AT Application in bytes.
  u32 StackSize // Open AT OS call stack area size in bytes.
  u32 HeapSize // Open AT OS  Heap area size in bytes.
  u32 GlobalSize //Open AT OS  Global variable area size in bytes.
} adl_memInfo_t
```

Returned values:

OK on success

ADL_RET_ERR_PARAM, if there is an error in the parameter.

7. Memory Limitations

The global variables, call stack and dynamic memory allocated are all part of the RAM allocated to Open AT Application. When an Open AT Application is developed on a embedded module B-type (with 128Kb of RAM), the whole RAM memory is not available for the Open AT Application as some of it is used by call stack and library ADL variables. The call stack uses at least 1KB of RAM. In addition to that 0.5KB of memory is used by the internal variables of the library.

8. Sample Code

```
#include "adl_global.h"
#if __OAT_API_VERSION__ >= 400
const u16 wm_apmCustomStackSize = 4096;
#else
u32 wm_apmCustomStack[1024];
const u16 wm_apmCustomStackSize = sizeof(wm_apmCustomStack);
#endif
void adl_main ( adl_InitType_e InitType )
{
        ascii * buffer;
        u32 total_allocation = 0;
        u16 max_size = 500;
        TRACE ((1, "Embedded Application: Main”));
        adl_atSendResponse(ADL_AT_UNS,"\r\nInside adl_main\r\n");
```

```
                    if (buffer = ((ascii *) adl_memGet(max_size)))
                    {
                              wm_memset (buffer,'0', max_size);
                              total_allocation = total_allocation + max_size;
                              TRACE ((1,"memory allocated is %d", total_allocation));
               adl_memRelease(buffer);       // releasing the memory
                    }
                    else
                    {
                              TRACE ((1,"unable to allocate memory"));
                    }
          }
```

Summary

The following points have been covered in this chapter

- **The memory can be allocated using the void * adl_memGet(u32 size) API.**
- **The adl_memGet API leads to an exception (memory error 0x10) if there is not enough RAM for the required size).**
- **The memory can be released using the bool adl_memRelease(void **ptr) API.**
- **The memory should be released only if it has been allocated previously, otherwise it will lead to an exception (RTK exception 155).**

CHAPTER 12

Debug Traces

1. Objective

This chapter introduces you to the concept of debug traces in an Open AT Application and describes the debug configuration settings of the Open AT OS.

2. Debug Traces in an Open AT Application

Debug traces can be used to return values of variables at runtime, making it easier to debug the Open AT Application. The software trace strings are displayed on the Trace View window of Developer Studio. There are several ways to embed these trace strings in an Open AT Application depending on the Open AT OS configuration settings. To use the debug traces in Open AT Application you must include the header file adl_traces.h in the Open AT Application.
The following sections of this chapter describe these configuration settings, and their APIs and Parmeters.

3. Build Configuration Macros

3.1. 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 Target Monitoring Tool at runtime).

3.2. Full Debug Configuration

When the Full Debug configuration is selected in the 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.

3.3. Release Configuration

When the Release configuration is selected in the 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 Target Monitoring Tool at runtime).

3.4. TRACE macro

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_ )
```

3.5. DUMP macro

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_ )
```

3.6. FULL TRACE macro

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. FULL DUMP macro

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_ )
```

4. Displaying the Debug Trace

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

- for tasks: on the trace element name defined in the tasks declaration table
- 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 the Target Monitoring Tool:
Value of I: 123

Prototype:

s8 adl_trcPrint (u8 Level, const ascii* strFormat, ...)

Parameters:

Level: This defines the trace level. The trace level range is from 1 to 32. Traces are displayed on the ADL level on the Trace View window of Developer Studio.

strFormat: This parameter is the trace string, which can use the standard C *sprintf* syntax. The maximum displayed string length is 256 bytes. If this string is longer it will be truncated on display.

...: 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.
'%s' character, normally used to insert strings, is not supported by the the trace function.
The trace display should be limited to 255 bytes. If the trace string is longer, it will be truncated.

5. Dumping the buffer

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

- for tasks: on the trace element name defined in the tasks declaration table
- 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 the maximum length of a display line 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 truncated 80-byte line will end with an ellipse (…).

An example follows:

```
u8 buffer [200], i;
for (i=0;i<200;i++)
buffer[i] = i;
DUMP (1, buffer, 200);
```

At runtime this code sample will display the following three lines on the CUS4 level1 on the Trace View window of Developer Studio:

```
00 01 02 03 04 05 06 07 08 09 [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: This defines the trace level. The trace level range is from 1 to 32. Traces are displayed on the ADL element on the Trace View window of Developer Studio.

DumpBuffer: This parameter is the address of the buffer to be dumped**.**
DumpLength: This parameter is the length of the buffer to be dumped.

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.

6. Sample Code

```
#include "adl_global.h"
#if __OAT_API_VERSION__ >= 400
const u16 wm_apmCustomStackSize = 4096;
#else
u32 wm_apmCustomStack[1024];
const u16 wm_apmCustomStackSize = sizeof(wm_apmCustomStack);
#endif
void adl_main ( adl_InitType_e InitType )
{
ascii * buffer;
        u32 total_allocation = 0;
        u16 max_size = 256;

        if (buffer = ((ascii *) adl_memGet(max_size)))
        {
                wm_memset (buffer,'$', max_size);
                total_allocation = total_allocation + max_size;
                TRACE ((1,"memory allocated is %d; Contents of buffer:",
 total_allocation));
                TRACE ((1, buffer ));
                TRACE ((1, "DUMP of contents of buffer (in hex):" ));
                DUMP (1, buffer, total_allocation);
    adl_memRelease(buffer);    /* releasing the memory */
        }
        else
        {
                TRACE ((1,"unable to allocate memory"));
        }
}
```

Summary

The following points have been covered in this chapter

- **Debug trace service is used to display the software trace strings on the Trace View window of Developer Studio.**
- **Debug traces are used in three configurations:**
- **Debug configuration**
- **Full debug configuration**
- **Release configuration**
- **The DUMP macro displays the contents of a buffer in a hexadecimal format on the Trace View window of Developer Studio.**
- **The TRACE macro is used to print the trace string using the standard C sprintf syntax.**

CHAPTER 13

Flash Memory

1. Objective

This chapter describes the flash memory and its usage in the embedded module. Various APIs are used in the Open AT OS to manage the flash memory. These API are used to subscribe, write, read, and erase the flash object in the flash memory.

2. Flash Memory

Flash memory refers to a memory chip that holds its content even when the power is switched off. Flash memory is derived from EEPROM, but it is less expensive and provides higher bit densities. In embedded module, the flash memory consists if various logical regions as mentioned in the below figure:

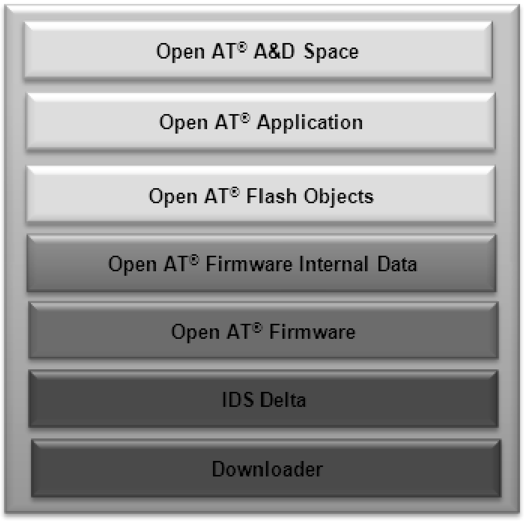


Figure 52 - Embedded module Flash

A part of the flash memory is reserved for flash object. This area is called "flash" in the Open AT OS memory mapping.
The objects are associated with a user-defined string handle. Each object can be uniquely identified by a numeric ID. The same is shown in the below mentioned figure:

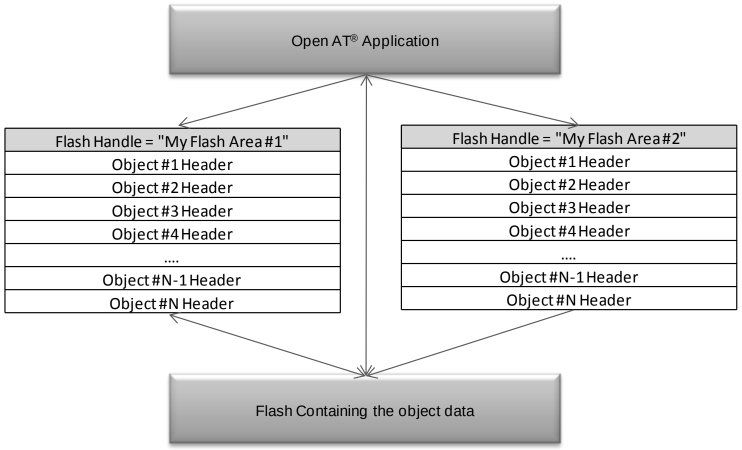


Figure 53 - Flash Object Architecture

A single object can use up to 30 Kbytes and only 2000 object identifiers can exist in Open AT Application at any point in time.

3. Flash APIs

This section describes the Open AT OS that you can use to manage the flash memory. These include APIs to create a flash handle, read from flash, write to flash, find object count and so on.

3.1. Subscribing to Flash Objects

Use the following API to subscribe to a set of objects that are identified by the handle.

Prototype:

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

Parameters:

Handle: This parameter is user-defined string used to identify a particular space on the flash memory. It is used by the other flash APIs to read and write on the flash memory.

NbObjectRes: This parameter specifies the number of objects or IDs related to the given handle. It also specifies that IDs are in the 0 to (NbObjectsRes-1) range.

Returned Values:

OK is returned on successful subscription of the flash memory.

ADL_RET_ERR_PARAM is returned in case of a parameter error.

ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the flash service is already subscribed.

ADL_FLH_RET_ERR_NO_ENOUGH_IDS is returned if there are not enough object IDs.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

NOTE :
It is sufficient to subscribe to the handle only once.
You cannot unsubscribe from a handle. To release the handle and object associated with it, you must issue the AT+WOPEN=3 command. The flash object of the embedded Open AT Application will be erased.
The object IDs and the handle associated with the flash subscription are stored in the flash memory when you subscribe to it.

3.2. Flash Object Exist Function

The following API checks whether a specified flash object exists in the flash memory.

Prototype:

```
s32 adl_flhExist (ascii * Handle, u16 ID)
```

Parameters:

Handle: This parameter is a user-defined string. It is used by the other flash APIs to read and write on the flash memory.

ID: This parameter is the ID of the flash object that you want to check for in the flash memory.

Returned Values:

If successful, this function returns the length of the specified flash object.

OK is returned if the flash object does not exist.

ADL_RET_ERR_UNKNOWN_HDL is returned if the handle is not subscribed.

ADL_FLH_RET_ERR_ID_OUT_OF_RANGE is returned if the ID is out of the handle range.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

3.3. Erasing the Flash Memory

Use the following API to erase the flash object of a given ID from the specified handle:

Prototype:

```
s8 adl_flhErase (ascii * Handle, u16 ID)
```

Parameters:

Handle: This parameter gives the handle of the subscribed set of objects.

ID: This parameter gives the ID of the flash object that you want to erase.

Returned Values:

OK is returned if the specified flash object is erased successfully.

ADL_RET_ERR_UNKNOWN_HDL is returned if handle is not subscribed.

ADL_FLH_RET_ERR_ID_OUT_OF_RANGE is returned if the ID is outside the handle range.

ADL_RET_ERR_FATAL is returned if a fatal error has occurred.

ADL_FLH_RET_ERR_OBJ_NOT_EXIST is returned if the object does not exist.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

3.4. Writing Objects to the Flash Memory

This function writes the flash object from the given Handle at the given Id for the length of the sting provided. A single flash object can use up to 30 Kbytes.
Use the following API to write flash objects to the flash memory.

Prototype:

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

Parameters:

Handle: This parameter gives the handle of the subscribed set of objects.

ID: This parameter gives the ID of the flash object that you want to write to the flash memory.

Len: This parameter gives the length of the flash object that you want to write to the flash memory.

WriteData: This parameter gives the string that you want to write to the flash object.

Returned Values:

OK is returned if the flash object is written successfully.

ADL_RET_ERR_PARAM is returned if a parameter has an incorrect value.

ADL_RET_ERR_UNKNOWN_HDL is returned if the handle is not subscribed.

ADL_FLH_RET_ERR_ID_OUT_OF_RANGE is returned if the ID is out of the handle range.

ADL_RET_ERR_FATAL is returned if a fatal error has occurred.

ADL_FLH_RET_ERR_MEM_FULL is returned if there is no more space in the flash memory.

ADL_FLH_RET_ERR_NO_ENOUGH_IDS is returned if the object cannot be created because of the ID number limitation.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

NOTE :
A single object can use up to 30 Kbytes and only 2000 object identifiers can exist at the same time.
If the power is cut off during a flash write sequence, the data that the current operation was writing will be lost. However, data that was previously written is retained and can be retrieved.

3.5. Reading from the Flash Memory

Use the following API to read the flash object from flash memory.

Prototype:

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

Parameters:

Handle: This parameter gives the handle of the subscribed set of objects.

ID: This parameter gives the ID of the flash object that you want to read.

Len: This parameter gives the length of the flash object that you want to read.

ReadData: This parameter gives the string allocated to store the read flash object.

Returned Values:

OK is returned on successful reading of the flash object.

ADL_RET_ERR_PARAM is returned if one parameter has an incorrect value.

ADL_RET_ERR_UNKNOWN_HDL is returned if the handle is not subscribed.

ADL_FLH_RET_ERR_ID_OUT_OF_RANGE is returned if the ID is out of the handle range.

ADL_RET_ERR_FATAL is returned if a fatal error has occurred.

ADL_FLH_RET_ERR_OBJ_NOT_EXIST is returned if the object does not exist.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

3.6. Remaining Flash Memory

Use the following API to get the size of remaining space in the flash memory.

Prototype:

```
u32 adl_flhGetFreeMem (void)
```

Parameters:

None.

Returned Values:

This API returns the size of the free space flash memory in bytes.

3.7. Handle ID Count

Use the following API to get the ID count associated with the given handle. If the handle is set to NULL, then this API lists the total ID count in the flash memory.

Prototype:

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

Parameters:

Handle: This parameter gives the handle of the subscribed set of objects.

Returned Values:

This API returns the ID count associated with the given handle on success. If the Handle parameter is set to NULL, then this API returns the total ID count in the flash memory.

ADL_RET_ERR_UNKNOWN_HDL is returned if the specified handle is not subscribed.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

3.8. Handle Memory Size

Use the following API to get the size of flash memory space used by the specified ID range and handle.

Prototype:

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

Parameters:

Handle: This parameter gives the handle of the subscribed set of objects. If this parameter is set to NULL, the total used size of the flash memory will be returned.

StartID: This parameter gives the starting ID of the range for which you want to find the size of the used memory space. The value must not exceed the EndID value.

EndID: This parameter gives the last ID of the range for which you want to find the size of the used memory space. This must be greater than the start ID.

Returned Values:

If this API is successful it returns the size of the used memory space associated with the given handle. If the Handle is set to NULL, then this API returns the total size of the used flash memory.

ADL_RET_ERR_PARAM is returned in case of parameter error.

ADL_RET_ERR_UNKNOWN_HDL is returned if the handle is not subscribed.

ADL_FLH_RET_ERR_ID_OUT_OF_RANGE is returned if the provided ID is out of the specified range.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

4. Sample Code

```
/* sample code implementing the flash API */

#include "adl_global.h"
#include "adl_flash.h"
const u16 wm_apmCustomStackSize = 4096;

ascii *hndle = "flash_handle";
u16 startId=0;
u16 count=0;
u8 *buffer_write;
u16 BlockSize = 3;

u8 *buffer_read;
/* Function to write data in Flash Memory*/
bool Flashwrite_Handler( adl_atCmdPreParser_t *paras )
{
  s8 ret_val;
  buffer_write = (u8 *) adl_memGet( BlockSize );
  wm_memset( buffer_write, 'A', BlockSize-1 );
  buffer_write[BlockSize]='\0';
  TRACE( ( 1, "Inside Flashwrite_Handler" ) );
  adl_atSendResponse(ADL_AT_RSP,"\r\nInside Flashwrite_Handler\n");
```

```
 ret_val = adl_flhWrite(hndle, startId, BlockSize, buffer_write);
 TRACE((1,"Return value of flash write is %d",ret_val));
 if( ret_val == OK )
 {
   adl_atSendResponse(ADL_AT_RSP,"\r\nDATA WRITTEN TO FLASH\n");
 }
 else
 {
   adl_atSendResponse(ADL_AT_RSP,"\r\nERROR IN WRITING FLASH\n");
 }
 adl_memRelease( buffer_write );
 return TRUE;
}

/* Function to delete the objects written into flash memory*/
bool Flashdelete_Handler( adl_atCmdPreParser_t *paras )
{
 s8 resp;
 TRACE( ( 1, "Inside Flashdelete_Handler" ) );
 adl_atSendResponse(ADL_AT_RSP,"\r\nfLASH DELETE HANDLER\n");
 resp = adl_flhErase(hndle, count);
 if( resp == OK )
 {
   adl_atSendResponse(ADL_AT_RSP,"\r\nFLASH DELETED\n");
 }
 else
 {
   adl_atSendResponse(ADL_AT_RSP,"\r\nERROR IN DELETING FLASH\n");
 }
 return TRUE;
}

 /* Function to read flash objects  */
bool Flashread_Handler( adl_atCmdPreParser_t *paras )
{
 s32 len=0;
 s8 ret_val;

 buffer_read = (u8 *) adl_memGet( BlockSize );

 TRACE( ( 1, "Inside Send_Flashget_Handler" ) );
 adl_atSendResponse(ADL_AT_RSP,"\r\nFLASH READ HANDLER\n");
 len = adl_flhExist(hndle, count);
 TRACE( ( 1, "Return value of adl_flhExist = %d", len ) );
 if ( len > 0 )
 {
   ret_val = adl_flhRead(hndle, count, len, buffer_read);
```

```
      TRACE((1,"Return value of adl_flhRead = %d",ret_val));
      if( ret_val == OK )
      {
        adl_atSendResponse(ADL_AT_RSP,"\r\nFLASH DATA READ:");
        adl_atSendResponse(ADL_AT_RSP,buffer_read);
        adl_atSendResponse(ADL_AT_RSP,"\n");
      }
      else
      {
        adl_atSendResponse(ADL_AT_RSP,"\r\nERROR IN READING FLASH\n");
      }
    }
    else if ( len == OK )
    {
      adl_atSendResponse(ADL_AT_RSP,"\r\nFLASH OBJ DOESNOT EXIST\n");
    }
    else
    {
      adl_atSendResponse(ADL_AT_RSP,
              "\r\nERROR WHILE CHECKING FOR FLASH EXISTANCE  \n");
    }
    adl_memRelease( buffer_read );
    return TRUE;
  }

  void adl_main ( adl_InitType_e InitType )
  {
    s8 ret_val1;
    TRACE (( 1, "Embedded Application: Main" ));
    adl_atCmdSubscribe("AT+FLASHREAD",(adl_atCmdHandler_t)Flashread_Handler,
                    ADL_CMD_TYPE_ACT);
    adl_atCmdSubscribe("AT+FLASHWRITE",(adl_atCmdHandler_t)Flashwrite_Handler,
                    ADL_CMD_TYPE_ACT);
    adl_atCmdSubscribe("AT+FLASHDELETE",(adl_atCmdHandler_t)Flashdelete_Handler,
                   ADL_CMD_TYPE_ACT);
    ret_val1= adl_flhSubscribe(hndle, 100);
    TRACE((1,"Return value of flash subscribe is %d",ret_val1));
  }
```

NOTE:
The handle needs to be subscribed only once.
The parameter which provides the length of the flash object (len) should exceed the actual length of the flash object
If the power is cut off during a flash write sequence, the data that the current operation was writing will be lost. However, the data that was previously written is retained and can be retrieved.

Summary

The following points have been covered in this chapter

- **Flash memory refers to a memory chip that holds its content without power.**
- **Flash memory is derived from EEPROM, but it is less expensive and provides higher bit densities.**
- **The flash APIs are declared in the adl_flash.h header file.**
- **To subscribe to the flash service, use the adl_flhSubscribe API.**
- **To check for the existence of the flash object, use the adl_flhExist API.**
- **To erase the flash object, use the adl_flhErase API.**
- **To write to the flash memory, use the adl_flhWrite API.**
- **To read from the flash memory, use the adl_flhRead API.**
- **To get the remaining flash memory size, use the adl_flhGetFreeMem API.**
- **To get the ID count for the given handle, use the adl_flhGetIDCount API.**
- **To get the size of the used memory space by a handle, use the adl_flhGetUsedSize API.**

CHAPTER 14

Application and Data Storage Service

1. Objective

This chapter introduces you to the Application and Data Storage Volume (A&D) Services and explains the APIs which can be used to manipulate this service. It also explains the concept behind the A&D storage memory and the process which should be followed to use this service. This chapter answers the following questions:

- What is A&D memory? What are its uses?
- How to subscribe to, and unsubscribe from the A&D service?
- How to write data in an A&D cell?
- How to install the application stored in the A&D storage?
- How to delete an A&D storage cell?
- How to manage the A&D storage effectively?

2. Application and Data Storage Memory

The DOTA I/II/III (Download over the Air) service is provided by Open AT OS to upgrade the existing/running embedded Open AT Applications and Firmware dynamically over the air. Open AT Application uses the special storage in flash known as the A&D storage to store the contents of the new Open AT Application/ Firmware that is downloaded. The new Open AT Application can be installed and executed in place of the old application. The old Open AT Application will be erased when the new Open AT Application is installed.

These operations are done over the air using a GSM/GPRS link and this reduces the cost of upgrading the software as the physical proximity with the embedded module is not required to upgrade/install an embedded Open AT Application. You do not need to visit the installation site of the embedded module to upgrade the software. You only need to implement the DOTA I/II/III functionality in your Open AT Application and send the new/ upgraded Open AT Application to your embedded module over the GPRS/GSM data link. Moreover, as the A&D storage is a part of the flash memory, the Open AT Application/ Firmware contents stored in this area are non-volatile. Hence, you can install the new Open AT Application/ Firmware whenever you require it. You do not have to worry about the risk of losing the contents of the new Open AT Application when the power to the embedded module is shut off.
Flash memory refers to a memory chip that holds its content even when the power is switched off. Flash memory is derived from EEPROM, but it is less expensive and provides higher bit densities.

A part of the flash memory is reserved for A&D storage space. This area is called "A&D space" in the Open AT OS memory mapping.

A cell is a basic unit of A&D storage that it is allocated for data. The size of a cell can be specified during allocation. This is a logical entity as compared to a sector, which is a physical unit.

A&D storage memory is a part of flash which can be used by the Open AT Application to store data objects or passive applications. Data objects can be any piece of information which you want to be stored permanently. Passive applications are inactive (not in execution) and are simply stored as raw data in the A&D storage. You can execute the Open AT Application as and when you want by using the adl_adInstall API described later in this chapter.

In Firmware, the command AT+WOPEN=6 can be used to configure the amount of memory to be allocated for A&D. The remaining memory will be available for Open AT Application binary. Please refer to the AT command interface guide for more details on this command.

3. Uses of A&D Storage

A&D storage can be used to store various kinds of information such as strings, EEPROM configuration files and so on. In addition, you can also store a compiled binary Open AT Application (.dwl file) in this storage. When required, you can initiate and run this Open AT Application. This can provide you multitude of features such as:

You can send a compiled Open AT Application (.dwl file) to the embedded module over an FTP connection or a GPRS/GSM data connection. The embedded module can store the Open AT Application in the A&D storage. Hence, you can update your old Open AT Application by sending new/updated applications to the embedded module. These Open AT Application are stored in the A&D storage memory. When required, you can execute this new Open AT Application by overwriting the old Open AT Application. In this way, you can update and execute the latest Open AT Application as and when needed. This process is also known as DOTA I (Download Over The Air).

A&D storage are could be used to store a .dwl file (that it could be a new embedded Open AT Application binary file, or an EEPROM configuration file, or an XModem downloader binary file). The dwl file stored can be installed whenever is required.

- You can download your Open AT Application to the A&D storage using the protocols like XModem over data call or FTP over GPRS or any other protocol.

- You can even download complete Firmware in the A&D storage cell also. The contents of the cell can then be installed and hence, a firmware upgrade can also be performed. This is called DOTA-II.

- You can even download delta of Firmware in the A&D storage cell also. The contents of the cell can then be installed and hence, a firmware upgrade can also be performed. This is called DOTA-III.

NOTE :
When an Open AT Application downloaded in the A&D storage is installed, the older application (if any) is erased, and the new application will take control when it is executed.

4. Subscribing to A&D Storage Services

The APIs related to the A&D storage service are declared in the adl_ad.h header file. Hence include this file to use the A&D storage services.

In order to avail the A&D services, your Open AT Application must subscribe to the A&D services using the following API:

Prototype:

s32 adl_adSubscribe (u32 CellID, u32 Size)

Parameters:

CellID: Each A&D cell is identified by a particular identifier. This parameter holds the cell identifier of the A&D space to which you want to subscribe. Here you can enter a cell ID which will uniquely identify the cell that you want to use. This cell may already exist. In this case, the cell will continue to hold the previously allocated space. However, in case, the cell does not exist, required cell is created and allotted the memory size requested by the second parameter.

Size: This parameter holds the size of cell (in bytes) to which you want to subscribe. If the cell exists, this parameter will be ignored. If you want a cell of variable size, then set this parameter to ADL_AD_SIZE_UNDEF.

Returned Values:

A positive or null handle is returned if the API is successful. This handle will then be used in other operations such as reading and writing in the subscribed cell.

ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the cell is already subscribed.

ADL_AD_RET_ERR_OVERFLOW is returned if there is not enough space for the allocation.

ADL_AD_RET_ERR_NOT_AVAILABLE is returned if there is no A&D space available on the product.

ADL_RET_ERR_PARAM is returned if cell ID parameter is 0xFFFFFFFF.

ADL_RET_ERR_BAD_STATE is returned if another undefined size cell is already subscribed and not finalized.

ADL_RET_ERR_SERVICE_LOCKED is returned if this API is called from low level interrupt handler.

5. Unsubscribing from A&D Storage Services

If you no longer require the A&D storage services, you can unsubscribe from these services using the following API

Prototype:

s32 adl_adUnsubscribe (s32 Handle)

Parameters:

Handle: Handle (value) returned by the adl_adSubscribe () API.

Returned Values:

OK is returned if A&D storage services are successfully unsubscribed.

ADL_RET_ERR_UNKNOWN_HDL is returned if the handle provided is invalid.

ADL_RET_ERR_SERVICE_LOCKED is returned if this API is called from low level interrupt handler.

6. A&D events

6.1. Subscribing to A&D events

This API is used to subscribe for all the events related to Application and data storage.

Prototype:

s32 adl_adEventSubscribe (adl_adEventHdlr_f Handler)

Parameters:

Handler: Call back function to notify all the event related to the application and data storage.

Prototype:

typedef void (*adl_adEventHdlr_f) (adl_Event_e Event, u32 Progress)

Parameter:

Event: Following event are notified in the handler.

ADL_AD_EVENT_FORMAT_INIT: API to format the A&D storage has been called.

ADL_AD_EVENT_FORMAT_PROGRESS: Format process is ongoing.

ADL_AD_EVENT_FORMAT_DONE: Format process for the A&D storage is completed.

ADL_AD_EVENT_RECOMPACT_INIT: Function to start the re-compaction of the A&D storage has been invoked.

ADL_AD_EVENT_RECOMPACT_PROGRESS: Re-compaction process is in progress.

ADL_AD_EVENT_RECOMPACT_DONE: Re-compaction process has been completed.

ADL_AD_EVENT_INSTALL: Install API has been called.

Progess: This parameter tells the status of a process.

Returned Values:

Positive or null is returned on success.

A negative can signify the following:

ADL_RET_ERR_PARAM is returned if parameter is invalid.

ADL_RET_ERR_NO_MORE_HANDLER is returned if A&D service has been subscribed for more than 128 times.

ADL_RET_ERR_SERVICE_LOCKED is returned if this API is called from low level interrupt handler.

6.2. Unsubscribing from A&D events

This API is used to unsubscribe the Application and data storage event service.

Prototype:

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

Parameters:

EventHandle: Handle returned during the time of subscription.

Returned Values:

OK is returned on success.

ADL_RET_ERR_UNKNOWN_HDL is returned if handle is unknown.

ADL_RET_ERR_NOT_SUBSCRIBED is returned if no A&D handler has been subscribed.

ADL_RET_ERR_BAD_STATE is returned if the format or re-compaction process is running with the given handle.

ADL_RET_ERR_SERVICE_LOCKED is returned if this API is called from low level interrupt handler.

6.3. Writing to a Subscribed A&D Storage Cell

Open AT OS provides an API to write to a subscribe A&D storage cell. The API has the following prototype:

Prototype:

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

Parameters:

Handle: Handle returned by the earlier call to adl_adSubscribe () API.

Size: Size of data buffer that is to be written to the cell.

Data: This is the pointer to the data to be written into the subscribed A&D storage cell.

Returned Values:

An OK is returned if the write operation is successful.

ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is not valid.

ADL_RET_ERR_PARAM is returned if a parameter is incorrect.

ADL_RET_ERR_BAD_STATE is returned if the cell is finalized.

ADL_AD_RET_ERR_OVERFLOW is returned if the write operation exceeded the allocated cell size.

ADL_RET_ERR_SERVICE_LOCKED is returned if this API is called from low level interrupt handler.

6.4. Reading from a Subscribed A&D Cell

You can get information related to a specified cell using the following API:

Prototype:

```
s32 adl_adInfo (s32 Handle, adl_adInfo_t *Info)
```

Parameters:

Handle: The A&D cell handle that is returned by the adl_adSubscribe () API.

Info: This is a pointer to a structure of type adl_adInfo_t which will contain the requested information pertaining to the cell identified by the Handle (first parameter), after the call is complete.

This structure has the following prototype:

```
typedef  struct
{
                u32 identifier;   //cell identifier
        u32 size;        //cell size
        void *data;      //pointer to the stored data
        u32        remaining;   //remaining writable spaces unless finalized.
        bool finalised;   //TRUE if cell is finalized, FALSE otherwise.
} adl_adInfo_t;
```

Returned Values:

An OK is returned if information retrieval for a particular cell is successful.

ADL_RET_ERR_PARAM is returned in the event of a parameter error.

ADL_RET_ERR_UNKNOWN_HDL is returned if the specified handle is invalid.

ADL_RET_ERR_BAD_STATE is returned if the required cell is a not finalized or an undefined size.

ADL_RET_ERR_SERVICE_LOCKED is returned if this API is called from low level interrupt handler.

6.5. Finalizing the A&D cell

After you have completed the write operation, you can write-protect the cell by finalizing it. Use the following API to finalize and protect the cell contents from modifications:

Prototype:

```
s32 adl_adFinalise (s32 Handle)
```

Parameters:

Handle: Handle (value) returned by the earlier call to adl_adSubscribe () API.

Returned Values:

An OK is returned on successful finalization of the specified cell.

ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is not valid.

ADL_RET_ERR_BAD_STATE is returned if the cell was already finalized.

ADL_RET_ERR_SERVICE_LOCKED is returned if this API is called from low level interrupt handler.

6.6. Installing an Open AT Application Written in the A&D Storage Space

As explained earlier in this chapter, you can use the adl_adInstall () API to store a compiled Open AT Application in the A&D storage memory as raw data, and install (execute) the Open AT Application when required. This API installs the contents of the requested cell. It should contain a .dwl file (that it could be the new Open AT Application binary file, a Firmware binary file, an EEPROM configuration file, or an XModem downloader binary file).

Prototype:

```
s32 adl_adInstall (s32 Handle)
```

Parameters:

Handle: A&D cell handle that is returned by the earlier call to adl_adSubscribe () API.

Returned Values:

A successful installation will reset the embedded module. When the embedded module restarts, the adl_InitType_e parameter of adl_main function is set to ADL_INIT_DOWNLOAD_SUCCESS. This acts as an indication that the installation process was successful. After the product reset, the newly installed application will start automatically.

ADL_INIT_DOWNLOAD_ERROR is returned if the .dwl file installation is unsuccessful.

ADL_RET_ERR_BAD_STATE is returned if the cell (from which the file must be installed) is not finalized.

ADL_RET_ERR_UNKNOWN_HDL is returned if the handle is not valid.

ADL_RET_ERR_SERVICE_LOCKED is returned if this API is called from low level interrupt handler.

6.7. Deleting an A&D Storage Cell

Open AT OS provides the following API which can be used to delete an A&D storage cell.

Prototype:

```
s32 adl_adDelete (s32 Handle)
```

Parameters:

Handle: A&D cell handle that is returned by the adl_adSubscribe () API.

Returned Values:

An OK is returned on successful deletion of the specified storage cell.

ADL_RET_ERR_UNKNOWN_HDL is returned if the specified handle is invalid.

ADL_RET_ERR_SERVICE_LOCKED is returned if this API is called from low level interrupt handler.

7. Managing A&D Storage Space

You can manage the A&D storage space by making effective use of the Open AT OS to:

- Re-compact the A&D storage space
- Get information about the state of the A&D storage space
- Get the allocated cell list for the entire A&D storage space

Format A&D area

7.1. Re-compaction of the A&D Storage Space

Re-compaction is the process which releases the space and ID of deleted cells. After re-compaction, the space and ID of deleted cells are made available for use again. This process is started automatically when the deleted A&D Storage Space exceeds 50 percent of the total A&D storage space. Alternatively, you can initiate this process by calling the following API:

Prototype:

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

Parameters:

EventHandle: This is the event handle that is previously returned by the adl_adEventSubscribe() API. The event handler that is associated with the A&D event subscription will receive all the events that is related to the re-compaction process.

Returned Values:

An OK is returned on successful completion of the recompaction process.

ADL_RET_ERR_BAD_STATE is returned if recompaction process is in process.

ADL_RET_ERR_UNKNOWN_HDL is returned if the handle is unknown.

ADL_RET_ERR_NOT_SUBSCRIBED is returned if the A&D event subscription is not done.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interruption handler.

ADL_AD_RET_ERR_NOT_AVAILABLE is returned if there is no A&D space available on the product.

7.2. Retrieving Information on the State of the A&D Storage Space

Information regarding the state of the A&D storage space includes details about the free space, deleted memory space, total memory space, the number of allocated objects and so on. Open AT OS provides you an API which you can use to get this information. This API has the following prototype:

Prototype:

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

Parameters:

State: Pointer to the information structure for the A&D state. The values in this structure are filled by adl_adGetState API. After calling this API, this structure will contain the values that will indicate the state information of A&D storage.

This structure has the following type:

```
typedef struct
{
u32 freemem;            //Free memory space
u32 deletedmem;         //Deleted memory size
u32 totalmem;           //Total A&D storage space
u16 numobjects;  //Number of allocated objects
u16 numdeleted; //Number of deleted objects
u8 pad;                 //This field is not used.
} adl_adState_t;
```

Returned Values:

An OK is returned when the data is retrieved successfully.

ADL_AD_RET_ERR_NOT_AVAILABLE is returned if no A&D storage space is available on the product.

ADL_AD_RET_ERR_NEED_RECOMPACT is returned if a power down or a reset occurred when a re-compaction process was running.

ADL_RET_ERR_PARAM is returned if an incorrect parameter is passed to this function.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interruption handler.

7.3. Information on Allocated Cells in the A&D Storage Space

Open AT OS provides you the following API which can be used to get a list of the currently allocated cells. This API has the following prototype:

Prototype:

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

Parameters:

CellList: A pointer to a list wm_lst_t variable. The adl_adGetCellList API updates this list with the identifiers of the allocated cells.

Returned Values:

An OK is on successful retrieval of cell information.

ADL_AD_RET_ERR_NOT_AVAILABLE is returned if no A&D space is available on the product.

ADL_RET_ERR_PARAM is returned on parameter error.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interruption handler.

NOTE :
The adl_adDelete () API marks the cell identified by its handle as 'free'. The memory occupied by the cell, is retrieved after the next recompaction process.
It is advisable to finalize a cell after writing to it. This is because, if you have subscribed to a cell of variable size (ADL_AD_SIZE_UNDEF), and do a recompaction without finalizing it, it will not work.
Each subsequent write operation will append the data to the existing (written) data. You can have a sequence of write operations and after you complete the write operation, you can finalize the cell using the adl_adFinalise () API.

7.4. Formatting A&D area

This API can be used to format the whole A&D area.

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.

ADL_AD_EVENT_FORMAT_INIT: This event is received just after the process is launched.
ADL_AD_EVENT_FORMAT_PROGRESS: This event is received several times, indicating the process progression.
ADL_AD_EVENT_FORMAT_DONE: This event is received once the process is done.

Returned Values:

OK is returned on success
.
ADL_AD_RET_NOT_AVAILBLE is returned if no A&D space is available.

ADL_RET_ERR_UNKNOWN_HDL is returned if the handle is unknown.

ADL_RET_ERR_NOT_SUBSCRIBED is returned if no A&D event handler has been subscribed.

ADL_RET_ERR_BAD_STATE is returned if there is at least one subscribed cell.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interruption handler.

NOTE:
1. This API should be used before A&D cell subscription.
2. As whole A&D area needs to be formatted, this API might take some time for processing.

7.5. Searching for a Cell

This API can be used to search for a cell ID between the 2 cell identifiers.

Prototype:

```
s32 adl_adFindInit ( u32 MinCellID, u32 MaxCellID, adl_adBrowse_t *BrowseInfo )
```

Parameters:

MinCellID: This parameter is the minimum cell value.
MaxCellID: This parameter is the maximum cell value.
BrowseInfo: This parameter is the browse information which will be used with the adl_adFindNext () API.

```
 typedef struct
{
  u32 hidden[4]; // Memory space necessary for cell information
} adl_adBrowse_t;
```

Returned Values:

OK is returned on success.

ADL_AD_RET_ERR_NOT_AVAILABLE is returned if A&D space is not available.

ADL_AD_RET_ERR_PARAM is returned if provided parameter is not correct.

7.6. Searching for a next Cell ID

This API can be used for searching a cell ID based on the browse information provided by adl_adFindInit () API.

Prototype:

```
s32 adl_adFindNext ( adl_adBrowse_t *BrowseInfo, u32 *CellId )
```

Parameters:

BrowseInfo: This parameter is the browse information returned by adl_adFindInit () API.
CellID: This parameter is the next cell ID found.

Returned Values:

OK is returned on success.

ADL_AD_RET_ERR_NOT_AVAILABLE is returned if A&D space is not available.

ADL_AD_RET_REACHED_END is returned if no more elements are there to enumerate.

7.7. Retrieving the result of adl_adInstall

This API can be used to retrieve the result of adl_adInstall.

Prototype:

```
s32 adl_adGetInstallResult ( void )
```

Returned Values:

OK is returned 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 recoveryADL_AD_RET_ERR_OAT_DEACTIVATED if the Open AT application was deactivated at start-up because of reset loops

7.8. Read cell on factory volume

This API can be used to read cell on 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:

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.

8. Sample Code

```
#include "adl_global.h"
#include "adl_ad.h"
const u16 wm_apmCustomStackSize = 4096;

s32 sEventHandle;
u32 gCellId = 1;
s32 gCellHandle = -1;

void SubscribeCell(  )
{
 u8 * mData;
 ascii Msg[250];
 u16 mDataLen=10;


 // Subscribe Cell
 gCellHandle = adl_adSubscribe( gCellId, ADL_AD_SIZE_UNDEF );

 // Verify Handle
 if ( gCellHandle < 0 )
 {
  wm_sprintf( Msg, "\r\nERROR: adl_adSubscribe --> FAILED\r\n" );
  TRACE (( 1, Msg ));
  adl_atSendResponse ( ADL_AT_RSP, Msg );
  return;
 }
 else
 {
  wm_sprintf( Msg, "\r\n adl_adSubscribe --> PASSED\r\n" );
  TRACE (( 1, Msg ));
  adl_atSendResponse ( ADL_AT_RSP, Msg );

  // Allocate Data Buffer
  mData = (u8 *)adl_memGet( (u16)mDataLen );
```

```
if( mData == NULL )
{
 wm_sprintf( Msg, "\r\nERROR: adl_memGet --> FAILED\r\n" );
 TRACE (( 1, Msg ));
 adl_atSendResponse ( ADL_AT_RSP, Msg );
 return;
}
// Fill Data Buffer
wm_memset( mData, 'A', mDataLen );
// Write All Data In Cell
if ( adl_adWrite( gCellHandle, mDataLen, mData ) != OK )
{
 wm_sprintf( Msg, "\r\nERROR: adl_adWrite --> FAILED\r\n" );
 TRACE (( 1, Msg ));
 adl_atSendResponse ( ADL_AT_RSP, Msg );
 return;
}
else
{
 wm_sprintf( Msg, "\r\n adl_adWrite --> PASSED\r\n" );
 TRACE (( 1, Msg ));
 adl_atSendResponse ( ADL_AT_RSP, Msg );

 // Finalise Cell
 if ( adl_adFinalise( gCellHandle ) != OK )
 {
  wm_sprintf( Msg, "\r\nERROR: adl_adFinalise --> FAILED\r\n" );
  TRACE (( 1, Msg ));
  adl_atSendResponse ( ADL_AT_RSP, Msg );
  return;
 }
 else
 {
  wm_sprintf( Msg, "\r\n adl_adFinalise --> PASSED\r\n" );
  TRACE (( 1, Msg ));
  adl_atSendResponse ( ADL_AT_RSP, Msg );
  // Delete Cell

  if ( adl_adDelete( gCellHandle ) != OK )
  {
   wm_sprintf( Msg, "\r\nERROR: adl_adDelete --> FAILED\r\n" );
   TRACE (( 1, Msg ));
   adl_atSendResponse ( ADL_AT_RSP, Msg );
   return;
  }
  else
  {
          wm_sprintf( Msg, "\r\n adl_adDelete --> PASSED\r\n" );
          TRACE (( 1, Msg ));
          adl_atSendResponse ( ADL_AT_RSP, Msg );

         }

        }
      }

    }
  }

  void adEventHandler ( adl_adEvent_e Event, u32 Progress )
  {
    switch( Event )
    {
      case ADL_AD_EVENT_FORMAT_INIT:
        TRACE (( 1, "ADL_AD_EVENT_FORMAT_INIT Event is received" ));
        break;
      case ADL_AD_EVENT_FORMAT_PROGRESS:
        TRACE (( 1, "ADL_AD_EVENT_FORMAT_PROGRESS Event is received" ));
        break;
      case ADL_AD_EVENT_FORMAT_DONE:
        TRACE (( 1, "ADL_AD_EVENT_FORMAT_DONE Event is received" ));
        break;
      case ADL_AD_EVENT_RECOMPACT_INIT:
        TRACE (( 1, "ADL_AD_EVENT_RECOMPACT_INIT Event is received" ));
        break;
      case ADL_AD_EVENT_RECOMPACT_PROGRESS:
        TRACE (( 1, "ADL_AD_EVENT_RECOMPACT_PROGRESS Event is received" ));
        break;
      case ADL_AD_EVENT_RECOMPACT_DONE:
        {
          ascii Msg[250];
          TRACE (( 1, "ADL_AD_EVENT_RECOMPACT_DONE Event is received" ));
          wm_sprintf( Msg, "\r\n adl_adRecompact --> PASSED\r\n" );
          TRACE (( 1, Msg ));
          adl_atSendResponse ( ADL_AT_RSP, Msg );
          SubscribeCell();
        }

        break;
      case ADL_AD_EVENT_INSTALL:
        TRACE (( 1, "ADL_AD_EVENT_INSTALL Event is received" ));
        break;
    }
  }
```

```
void ReCompact( u8 Id, void *context )
{
 s32 sRet;

 ascii Msg[250];
 adl_atSendResponse ( ADL_AT_UNS, "\r\nRECOMPACT...\r\n" );


 sRet=adl_adRecompact( sEventHandle );
 if ( sRet != OK )
 {
  wm_sprintf( Msg, "\r\nERROR: adl_adRecompact  --> FAILED returns %d\r\n", sRet );
  TRACE (( 1, Msg ));
  adl_atSendResponse ( ADL_AT_RSP, Msg );
 }
}

void Clean()
{
 s32 sReturn;
 s8 MyIndex;
 ascii Msg[250];
 u32 * CellID;
 s32 CellHandle;
 u16 mLstCount = 0;
 u16 i;
 wm_lst_t * mCellList = adl_memGet( sizeof( wm_lst_t ) );
 adl_atSendResponse ( ADL_AT_UNS, "\r\nCLEAN...\r\n" );

 // Get Cell List
 sReturn=adl_adGetCellList( mCellList );
 if ( sReturn!= OK)
 {
  wm_sprintf( Msg, "\r\nERROR: adl_GetList --> FAILED returns %d\r\n", sReturn);
  TRACE (( 1, Msg ));
  adl_atSendResponse ( ADL_AT_RSP, Msg );
  return;
 }
 else
 {
  // Get Number Of Cell In List
  mLstCount = wm_lstGetCount( *mCellList );
  wm_sprintf( Msg, "\r\nList contain %d Cells\r\n\r\n", mLstCount );
  MyIndex=0;
  while (MyIndex<mLstCount)
  {
      wm_sprintf( Msg, "\r\n*************** List item %d Cells item:%d\r\n\r\n",wm_lstGetItem(mCellList, MyIndex) ,
MyIndex);
      adl_atSendResponse ( ADL_AT_RSP, Msg );
      MyIndex++;
    }

    TRACE (( 1, Msg ));
    adl_atSendResponse ( ADL_AT_RSP, Msg );

    // Loop until number of Cell is reached
    for ( i = 0; i < mLstCount; i++ )
    {
      // Get Cell ID From List
      CellID = (u32 *)wm_lstGetItem( *mCellList, i );
      if ( CellID != (u32 *)NULL )
      {
        wm_sprintf( Msg, "\r\nList ID %d Cell Pointer 0x%x ID %d\r\n\r\n", i, CellID, *CellID );
        TRACE (( 1, Msg ));
        adl_atSendResponse ( ADL_AT_RSP, Msg );
      }
      else
      {
        wm_sprintf( Msg, "\r\nInvalid Pointer 0x%x\r\n\r\n", i, *CellID );
        TRACE (( 1, Msg ));
        adl_atSendResponse ( ADL_AT_RSP, Msg );
        return;
      }

      // Allocate ti Cell Id to delete Cell
      CellHandle = adl_adSubscribe( *CellID, ADL_AD_SIZE_UNDEF );
      if (  CellHandle < 0 )
      {
        wm_sprintf( Msg, "\r\nERROR: adl_adSubscribe --> FAILED\r\n" );
        TRACE (( 1, Msg ));
        adl_atSendResponse ( ADL_AT_RSP, Msg );
        return;
      }
      else
      {
        if ( adl_adDelete( CellHandle ) != OK )
        {
          wm_sprintf( Msg, "\r\nERROR: adl_adDelete --> FAILED\r\n" );
          TRACE (( 1, Msg ));
          adl_atSendResponse ( ADL_AT_RSP, Msg );
          return;
        }
        else
        {
          wm_sprintf( Msg, "\r\nList ID %d Delete Cell Handle %x --> PASSED\r\n\r\n", i, CellHandle );
          TRACE (( 1, Msg ));
          adl_atSendResponse ( ADL_AT_RSP, Msg );
        }
      }
    }

    // Launch Timer for Clean
    adl_tmrSubscribe ( FALSE, 10, ADL_TMR_TYPE_100MS, ReCompact );
  }

  adl_memRelease( mCellList );
}

void adl_main ( adl_InitType_e InitType )
{
  ascii Msg[250];
  TRACE (( 1, "Embedded Application: Main" ));

  sEventHandle =  adl_adEventSubscribe(adEventHandler);
  if ( sEventHandle<0)
  {
    wm_sprintf( Msg, "\r\nERROR: sEventHandle  --> FAILED returns %d\r\n", sEventHandle);
    TRACE (( 1, Msg ));
    adl_atSendResponse ( ADL_AT_RSP, Msg );
    return;
  }
  else
  {
    wm_sprintf( Msg, "\r\nsEventHandle is OK\r\n" );
    TRACE (( 1, Msg ));
    adl_atSendResponse ( ADL_AT_RSP, Msg );
  }

  Clean();
}
          if (adl_adDelete (cell_handle)!=OK)
          {
                    TRACE((1,"Cannot delete the cell"));
          }
          else
                                                  {
                    TRACE((1,"Cell deleted successfully"));
          }
```

```
            if (adl_adRecompact(recompact_handler)!=OK)                                  {
                                    TRACE((1,"Cannot recompact"));
            }
            else
                                    {
                TRACE((1,"Recompaction request successful"));
            }
        }
```

Summary

The following points have been covered in this chapter

- **Open AT OS provides APIs to manage the A&D storage space. The A&D storage space can be used to store data objects or Open AT Application binary files (.dwl) files.**
- **You can install the Open AT Application stored in the A&D storage space. The previous application will be deleted in this case and the new application will be executed.**
- **The adl_adSubscribe () API can be used to subscribe to A&D storage services. Use the adl_adUnsubscribe () API to unsubscribe from the A&D storage services.**
- **To write to a specific cell use the adl_adWrite () API. To get the data stored in a particular cell, use the adl_adInfo () API.**
- **Data written to a cell should be finalized to make it read only. This can be done using the adl_adFinalise () API.**
- **To delete a cell, use the adl_adDelete () API. However this API only marks the cell for deletion. The storage space is freed only after the recompaction process.**
- **To recompact the A&D storage space, use the adl_adRecompact () API. The recompaction process is also automatically initiated if the deleted A&D Storage Space exceeds 50 percent of the total available A&D space.**
- **The adl_adInstall () API can be used to install the contents of a cell that contains a .dwl file. After successful installation, the embedded module will be restarted.**
- **The adl_adGetState () API can be used to get information about the state of the A&D storage space. This information includes information such as the total A&D memory space, free A&D memory space and the number of objects. The adl_adInfo () API can be used to get the information pertaining to a particular cell.**
- **The adl_adFindInit () and adl_adFindNext () are used for searching cell ID.**

CHAPTER 15

Secured Data Storage Service

1. Objective

This chapter describes the secured data storage service and its usage in the embedded module.

2. Introduction

The data which is stored in the flash of the embedded module can be read by any user. Hence it is not safe to store the confidential information in the flash. To provide security for the data stored in the flash, secured data storage is provided.

Secured data storage is a part of flash memory where the data will be stored in the ciphered format thus providing the security for the data. This is useful in case of storing confidential information such as password, user name etc.

3. Configuring Security Feature in Embedded module

Security Feature needs to be enabled before it could be used in the embedded module.
For enabling the feature, use of the dongle is mandatory because this feature requires a modification of the internal EEPROM through dwlwin.

Using the dongle .wpk executable needs to be downloaded into the embedded module.

Set/Enable the security features Option will be popped up. If selected, this option allows the user to load a secured key into the firmware (SDS Secured Data Storage).
Followed by this option, resizing of A&D storage and flash size, firmware download will be popped up. Select accordingly and proceed further.
Once the configuration and the firmware download is complete "Process Successful" message will be seen in dwlwin.
User can exit from the dwlwin and can start using SDS feature.

For more details please refer to the pdf "AirPrime WMP100 Development kit - Software download tutorial - 2009-02-11[1].pdf".

NOTE : The maximum size for the secured data service can be configured by DWLWIN while initially configuring the module (as stated in 15.3). Hence for a given module the "maximum SDS service size" is the size defined by the user during initial configuration. This size would be part of the whole flash available to the module (the flash available to the FW and not Open AT flash service).
The minimum to maximum size of the secured data varies from 0 to "maximum SDS service size".

4. Secured Data Storage APIs

ADL provides a secured data storage service to:

- Read/write/query data stored in ciphered format in non volatile memory (embedded module Flash).
- Update cryptographic keys to block replay/re-download attacks.

The APIs related to the secured data storage service is declared in the adl_sds.h header file.

4.1. Retrieving Information about the Secured Data Storage

This API is used to retrieve information about:

- Free space available for secured entries.
- Total space allocated for secured entries.
- Total number of secured entries.
- Maximum number of secured entries.
- Maximum size of one secured entry.

Prototype:

```
s32 adl_sdsStats (adl_sdsStats_t* Stats)
```

Parameters:

Stats: This parameter is a pointer to a adl_sdsStats_t which stores information about the secured data storage. The structure adl_sdsStats_t is defined as:

```
typedef  struct
{
  u32 MaxEntrySpace;  //Maximum size of one secured entry
  u32 FreeSpace;        //Available space for secured entries
  u32 TotalSpace ;       //Total space allocated for secured entries
  u16 MaxEntry ;         //Maximum number of secured entries
  u16 EntryCount ;      //Total number of secured entries
}adl_sdsStats_t;
```

Returned Values:

OK is returned on successful retrieval of secured data storage information.

ADL_RET_ERR_PARAM is returned if incorrect parameter is specified.

ADL_RET_ERR_NOT_SUPPORTED is returned if operation is not supported.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

4.2. Storing Secured Data

This API is used to store data in a ciphered format with the specific numeric ID. Data is updated if ID is already present otherwise new entry will be created. The updated data is ciphered before it is stored in the non-volatile memory.

Prototype:

```
s32 adl_sdsWrite (u32 ID, u32 Length, void* Source)
```

Parameters:

ID: This parameter is the numeric ID of the entry which ranges from 0 to maximum entry retrieved by adl_sdsStats API.

Length: This parameter indicates the length of the data. The maximum length of the data can be retrieved by adl_sdsStats API.

Source: This parameter is the pointer to data which needs to be stored.

Returned Values:

OK is returned on successful retrieval of secured data storage information.

ADL_RET_ERR_PARAM is returned if incorrect parameter is specified.

ADL_RET_ERR_MEM_FULL is returned if memory is less for storing the data.

ADL_RET_ERR_NOT_SUPPORTED is returned if operation is not supported.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

4.3. Reading Secured Data

This API is used to read data from a secured entry. The entry to be read is specified by the numeric ID.

Prototype:

s32 adl_sdsRead (u32 ID, u32 offset, u32 Length, void* Destination)

Parameters:

ID: This parameter is the numeric ID of the entry which ranges from 0 to maximum entry retrieved by adl_sdsStats API.

Offset: This parameter indicates the offset in the secured entry specified in terms of bytes. This parameter allows retrieving part of the entry.

Length: This parameter indicates the length of the data. The maximum length of the required entry should be retrieved by adl_sdsQuery API.

Destination: This parameter is the pointer to buffer where the retrieved data will be stored.

Returned Values:

OK is returned on successful retrieval of secured data storage information.

ADL_RET_ERR_PARAM is returned if incorrect parameter is specified.

ADL_RET_ERR_ENTRY_NOT_EXIST is returned if data entry doesn't exist.

ADL_RET_ERR_NOT_SUPPORTED is returned if operation is not supported.
ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

4.4. Retrieving Secured Data Entry Size

This API is used to check whether the secured data entry exists or not. If exists, it retrieves the size of the entry.

Prototype:

s32 adl_sdsQuery (u32 ID, u32* Length)

Parameters:

ID: This parameter is the numeric ID of the entry which ranges from 0 to maximum entry retrieved by adl_sdsStats API.

Length: This is a pointer which holds the length of the entry. This parameter can be set to NULL.

Returned Values:

OK is returned on successful retrieval of secured data storage information.

ADL_RET_ERR_PARAM is returned if incorrect parameter is specified.

ADL_RET_ERR_ENTRY_NOT_EXIST is returned if data entry doesn't exist.

ADL_RET_ERR_NOT_SUPPORTED is returned if operation is not supported.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

4.5. Deleting Secured Data Entry

This API is used to delete secured data entries.

Prototype:

```
s32 adl_sdsDelete (u32 ID)
```

Parameters:

ID: This parameter is the numeric ID of the entry which ranges from 0 to maximum entry retrieved by adl_sdsStats API.

Returned Values:

OK is returned on successful retrieval of secured data storage information.

ADL_RET_ERR_PARAM is returned if incorrect parameter is specified.

ADL_RET_ERR_ENTRY_NOT_EXIST is returned if data entry doesn't exist.

ADL_RET_ERR_NOT_SUPPORTED is returned if operation is not supported.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

4.6. Updating Cryptographic Keys

This API is used to re-generate the internal cryptographic keys. All the stored data remain available and still readable by application even after re-generation of keys.

When the internal keys are updated, the data available in the flash memory is re-encrypted according to new keys. Hence it affects the life span of non-volatile memory. The re-generation of keys should be done to provide more security to data. For e.g. If someone is able to decode the key, it will cause a security issue to the data if the key is not updated periodically.

Prototype:

```
s32 adl_sdsUpdateKeys (void)
```

Parameters:

None.

Returned Values:

OK is returned on successful retrieval of secured data storage information.

ADL_RET_ERR_NOT_SUPPORTED is returned if operation is not supported.

ADL_RET_ERR_FATAL is returned if it is not possible to write EEPROM.

ADL_RET_ERR_SERVICE_LOCKED is returned if called from low level interrupt handler.

NOTE :
The above 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.

5. Sample Code

```
#include "adl_global.h"
#include "adl_sds.h"
const u16 wm_apmCustomStackSize = 4096;

adl_tmr_t *timer_ptr;
u16 timeout_period = 5;
static counter=0;

void adl_main (adl_InitType_e adlInitType)
{
  s32 COUNTER_ID = 0;
  s32 ret = 0;
  u32 data_write=10;
  u32 data_read = 0;
  u32 size;
  u32 offset=0;
  TRACE((1,"Inside adl_main"));

  //Write the counter value
  ret = adl_sdsWrite( COUNTER_ID, sizeof(u32), &data_write );
  TRACE((1,"Return value of adl_sdsWrite = %d", ret));
  //Check whether entry exist or not
  ret = adl_sdsQuery( COUNTER_ID, &size );
  TRACE((1,"Return value of adl_sdsQuery = %d", ret));
  //Read the counter value
```

```
 ret = adl_sdsRead( COUNTER_ID, offset, size, &data_read );
 TRACE((1,"Return value of adl_sdsRead = %d", ret));
 TRACE((1,"Data Read= %d", data_read));
 //Decrement the counter value
 data_write--;
 //Write the counter value
 adl_sdsWrite( COUNTER_ID, size, &data_write );
 TRACE((1,"Return value of adl_sdsWrite = %d", ret));
 //Read the counter value
 adl_sdsRead( COUNTER_ID, offset, sizeof(u32), &data_read );
 TRACE((1,"Return value of adl_sdsRead = %d", ret));
 TRACE((1,"Data Read= %d", data_read));
 // delete entry
 ret = adl_sdsDelete( COUNTER_ID );
 TRACE((1,"Return value of adl_sdsDelete = %d", ret));
}
```

Summary

The following points have been covered in this chapter

- **Open AT OS provides APIs for reading/writing secured data.**
- **The adl_sdsStats API should be used to retrieve information about free space available for secured entries, total space allocated for secured entries, total number of secured entries, maximum number of secured entries and maximum size of one secured entry.**
- **The adl_sdsWrite API should be used to store data in a ciphered format with the specific numeric ID.**
- **The adl_sdsRead API should be used to read data from a secured entry.**
- **The adl_sdsQuery API should be used to check whether the secured data entry exists or not.**
- **The adl_sdsDelete API should be used to delete secured data entries.**
- **The adl_sdsUpdateKeys API should be used to re-generate the internal cryptographic keys.**

CHAPTER 16

Flow Control Management

1. Objective

This chapter introduces the Flow Control Manager (FCM). It explains the data and command flows in the embedded module and how these can be managed from within the environment.

2. Flow Control Manager

The FCM controls the flow of data between the embedded Open AT Application and the input/output (I/O) flow points. The I/O flows include the V24 serial link (UART1 and UART2), USB and the GPRS and GSM flows.

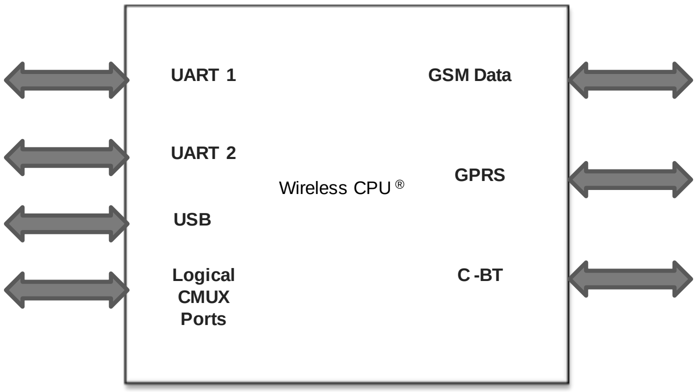


Figure 54 - Input Output Flows

The embedded Open AT Application can subscribe and unsubscribe to the different I/O flows. The embedded Open AT Application should subscribe to the V24 serial flow, in order to exchange data with the external application through the serial link. In order to send and receive data through the GSM or GPRS data channel, the embedded Open AT Application should subscribe to the GSM or GPRS flow.
There's no correlation between V24 and GSM or V24 and GPRS. Each flow can be subscribed independently without any specific order.

2.1. V24 Serial

In the V24 serial link, the Open AT Application can send and receive data as well as AT commands. The AT commands are sent from the external application to the embedded module and the Embedded module gives the responses back to the external application. Data could be transferred through the V24 serial flow between the external application and the embedded module.

V24 serial flow can subscribed with the serial link state switched to AT mode or data mode.

2.2. GSM and GPRS Flows

When the embedded Open AT Application does not subscribe to the GSM/GPRS flows, during a GSM/GPRS data call, the data are sent from the network directly to the V24 serial link. The embedded Open AT Application can subscribe to the GSM/GPRS flows in order to send and receive data. The data that are sent over GSM are raw data, but data that are sent over GPRS must be sent in the form of IP packets.

Following are the flows:

CMUX logical ports

- These ports are opened over physical ports such as UART1/UART2/USB.
- These flows are based on the 3GPP TS 27.010 multiplexing protocol.
- Only 4 logical ports can be opened at a time.

Bluetooth port

- These ports are independent of physical ports
- These ports are created when Bluetooth stack in started using Open AT OS APIs.

3. Flow Control Architecture

The Flow Control Manager Architecture figure shows the default behavior of the embedded module. The embedded module transmits all data from the serial link to the GSM data or the GPRS communication point.

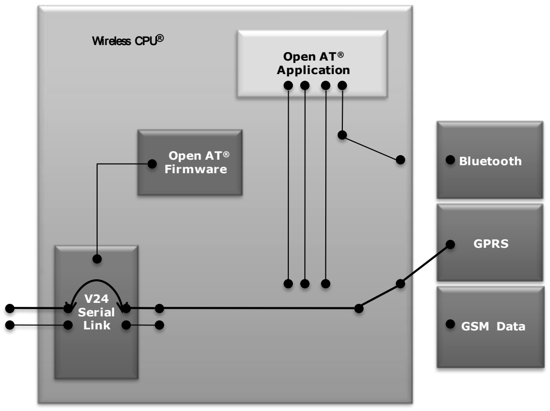


Figure 55 - Default State of Flow Control

An Open AT Application can subscribe to any flow (V24, GSM DATA or GPRS) to exchange data with the corresponding flow points. For example, if the embedded Open AT Application subscribes to the V24 flow then all the data from the V24 flow will be sent directly to the Open AT Application.

The different flow types that you can use are described in the sections that follow.

3.1. V24/USB Serial Flow in Data Mode from/to an Open AT Application

In this flow, the data is transmitted through the serial point (UART1, UART2) between the embedded Open AT Application and the external application. This happens when the Open AT Application has subscribed for the V24 flow and the serial link is switched in data mode. The flow of the data is shown in the Serial link Switched in Data Mode image.

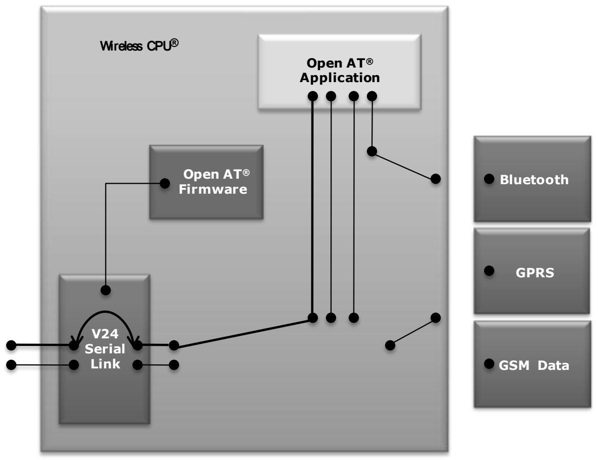


Figure 56- Serial Link Switched to the Data Mode

3.2. V24/USB Serial Link Switched in the AT Mode

When the serial link is switched to the AT mode, the AT commands are sent by an external application to the embedded module. The AT layer processes the AT commands and forwards the custom AT commands to the Open AT Application (if the Open AT Application has subscribed for the same).

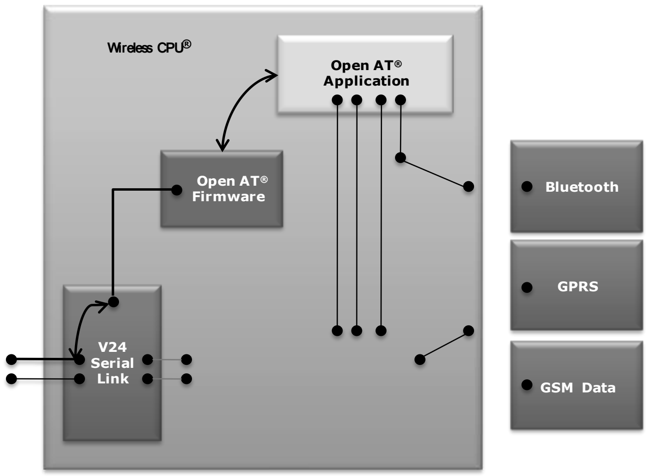


Figure 57 - Serial Link Switched to AT Mode

3.3. GSM and GPRS Data Flow from an Open AT Application

The Open AT Application can send and receive data over the GSM flow and GPRS flow. The Open AT Application subscribes to any of the GSM/GPRS flows and sends the data directly to the GSM/GPRS flow. In the case of GSM, the data can be sent after the GSM data or fax call is established. In the case of GPRS the data can be sent once the GPRS session is established.

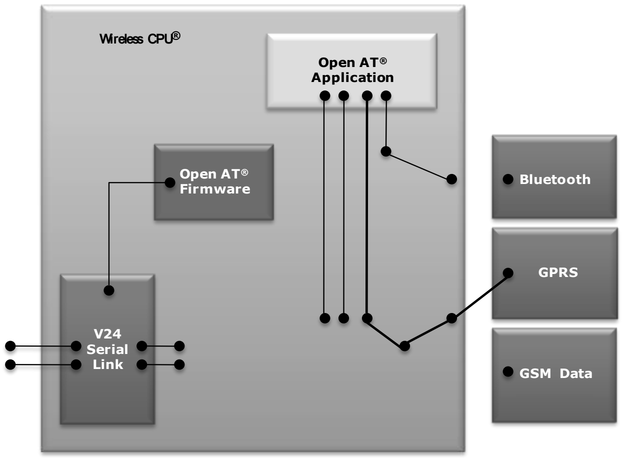


Figure 58 - GSM/GPRS Flow from an Open AT Application

4. FCM Service Provided by ADL Library

The ADL library provides several APIs for controlling the IO flows. Using these APIs the embedded Open AT Application can send and receive the data to and from the IO flows.
The FCM APIs are provided by the ADL library for the subscription of different Input Output flows. The APIs related to the secured FCM service is declared in the adl_fcm.h header file.

4.1. FCM Flows and Subscription

The Open AT Application must subscribe to an IO flow, before the flow can be made available to the Open AT Application. Use the adl_fcmSubscribe function to subscribe to a flow. This function takes the callback handlers for control and data events as parameters and returns a handle which can be used in all further FCM operations.

Outgoing data can be sent to the different flow points using the adl_fcmSendData function. Incoming data is processed by the data handlers specified in the adl_fcmSubscribe command.
There are different flows that can be opened: GSM data flow, GPRS flow, v24 UART1 flow, v24 UART2 flow and USB flow. Each flow is independent and it can be subscribed and unsubscribed separately.

The v24 flow can function in two different modes by switching the V24 serial link:

- AT mode
- Data mode

The AT mode is used to transmit AT commands to the embedded module. The Data mode is used to transfer data from the external application to the Open AT Application. The adl_fcmSwitchV24State function can be used to switch between these two modes.

The V24 flow can be made on UART1, UART2 or USB. A V24 Master flow subscription is used to manage the V24 flow. Only a Master subscriber can switch modes. The corresponding data handler is then intimated of mode switch events. The ADL_FCM_FLOW_V24_MASTER corresponds to ADL_FCM_FLOW_V24_UART1 and ADL_FCM_FLOW_V24 corresponds to ADL_FCM_FLOW_V24_UART1 | ADL_FCM_FLOW_SLAVE.

When the data handlers are called, they can indicate that they have processed the data by returning TRUE. This process is known as *releasing credit*. The data is then cleared in the V24 buffers so that the external application can continue sending data. If the data handler returns FALSE (credit is not released), then the buffers are not flushed. When the buffer is filled and can accept no more data, the V24 layer uses the hardware flow controls to stop the external application from sending more data.

The adl_fcmSubscribe function is used for the subscription of the different IO flow points. This API is used for the subscription of any of the IO flows. On successful subscription of a flow, the event ADL_FCM_EVENT_FLOW_OPENNED is received in the control handler. The function returns a handler which must be provided to other FCM service APIs. You can subscribe to the V24 flow multiple times, only after you subscribe to the V24 Master flow. The first V24 flow is subscribed in master mode and the others V24 flows are subscribed in slave mode.

The prototype of this API gives the definition of the API along with the parameters that need to be passed.
s8 adl_fcmSubscribe (adl_fcmFlow_e Flow, adl_fcmCtrlHdlr_f CtrlHandler, adl_fcmDataHdlr_f DataHandler)

Parameters:

Flow: The flow parameter decides the type of flow that needs to be subscribed. The allowed values of the flow are:

ADL_FCM_FLOW_GSM_DATA,

ADL_FCM_FLOW_GPRS,

ADL_FCM_FLOW_V24_UART1

ADL_FCM_FLOW_V24_UART2

ADL_FCM_FLOW_USB

For serial-link related flows (ADL_FCM_FLOW_V24_UART1 & 2), the corresponding UART has to be opened first with the "AT+WMFM" command, otherwise the subscription will fail. By default, only the UART1 is opened.

CtrlHandler: FCM control events handler, using the following type:

```
typedef bool (* adl_fcmCtrlHdlr_f) ( u8 event )
```

ADL provides FCM services to handle FCM events. The FCM flow is driven by several events received from the ADL layer. Based on these events the Open AT Application is aware of the current status of the subscribed flow.

The following table lists the flow events and their descriptions:

FCM Events	Description
ADL_FCM_EVENT_FLOW_OPENNED	This event is generated when the subscribed flow is opened.
ADL_FCM_EVENT_FLOW_CLOSED	This event is generated when the subscribed flow is closed.
ADL_FCM_EVENT_V24_DATA_MODE	This event is generated when the V24 Master is subscribed and Data mode is selected.
ADL_FCM_EVENT_V24_DATA_MODE_EXT	This event is generated when the Data mode switches from the external application.
ADL_FCM_EVENT_V24_AT_MODE	This event is generated when the V24 Master is subscribed and Data mode is selected.
ADL_FCM_EVENT_V24_AT_MODE_EXT	This event is generated when the AT mode switches from the external application.
ADL_FCM_EVENT_RESUME	This event is related to adl_fcmSendData() API.
ADL_FCM_EVENT_MEM_RELEASE	This event is related to adl_fcmSendData() API.

NOTE :
The flow control is activated on the V24 serial link, in the command (offline) mode. Payload data is located on the seven least significant bits (LSB) of every byte. So in every byte that is sent along the serial link, only the least seven bits contain the actual data. The rest of the data can be ignored.

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

DataHandler: FCM data 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 the next time the handler returns TRUE. On the V24 flow, all the subscribed data handlers are notified with a data event, and the credit will be released only if all handlers return TRUE as the default value.

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

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

- On serial link flows (UART physical & logical based ports): 120 bytes;
-
- On GSM CSD data port: 270 bytes;
-
- On GPRS port: 1500 bytes;

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.

When the serial link is in the Data mode, if the external application sends the sequence 1s delay; +++; 1s delay, the serial link is switched to the AT mode, and the V24 Master handler is notified by the ADL_FCM_EVENT_V24_AT_MODE_EXT event. The behavior then depends on the returned value. If it is TRUE, then all the V24 handlers are also notified with this event. The V24 MASTER handle cannot be unsubscribed in this state. If the returned value is FALSE, then the V24 handlers are not notified with this event, and the serial link is switched back to the Data mode immediately. In the first case, after the ADL_FCM_EVENT_V24_AT_MODE_EXT event, the V24 Master 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. The event is sent when the external application sends the ATO command; the serial link is switched to the Data mode, and all the V24 clients are then notified.

The following table lists the FCM flows and their descriptions:

FCM Flows	Description
ADL_FCM_FLOW_GSM_DATA	Subscribe to this flow to receive and send data via GSM data flow.
ADL_FCM_FLOW_GPRS	Subscribe to this flow to receive and send data via GPRS.
ADL_FCM_FLOW_V24_UART1	Subscribe to this flow to receive and send data via UART1. This is the same as ADL_FCM_FLOW_V24_MASTER.
ADL_FCM_FLOW_V24_UART2	Subscribe to this flow to receive and send data via UART2.
ADL_FCM_FLOW_USB	Subscribe to USB flow to send/receive data. Subject to availability

Returned Values:

A positive or null handle is returned on success. The Control handler will also receive a

ADL_FCM_EVENT_FLOW_OPENNED event when flow is ready to process.

ADL_RET_ERR_PARAM is returned if one parameter has an incorrect value.

ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the flow is already subscribed in master mode.

ADL_RET_ERR_NOT_SUBSCRIBED is returned if a slave subscription is made when master flow is not subscribed.

ADL_FCM_RET_ERROR_GSM_GPRS_ALREADY_OPENNED is returned if a GSM or GPRS subscription is made when the other one is already subscribed.

ADL_RET_ERR_BAD_STATE is returned if the required port is not available.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level Interrupt handler.

4.2. Unsubscribing FCM Flow

Use this function to unsubscribe from a subscribed FCM service, and to close the opened flows. The un-subscription will be effective only when the control event handler has received the ADL_FCM_EVENT_FLOW_CLOSED event. On the V24 flow, only the V24 MASTER Subscriber can close the flow. When other V24 subscribers unsubscribe, the operation will only free the related handle.
The prototype of this API gives the definition of the API along with the parameters that need to be passed.

Prototype:

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

Parameters:

Handle: Handle returned by the adl_fcmSubscribe function.

Returned Values:

OK on successful subscription of the IO flow. The control handler will also receive ADL_FCM_EVENT_FLOW_CLOSED event when flow is closed.

ADL_RET_ERR_UNKNOWN_HDL is returned if the handle is incorrect.

ADL_RET_ERR_NOT_SUBSCRIBED is returned if the flow is already unsubscribed.

ADL_RET_ERR_BAD_STATE is returned if the serial link is not in the AT mode. A negative handle is returned on failure.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.

4.3. Releasing FCM Credits

This function releases the credits for the requested flow handle. The V24 Master can handle this API. The Slave subscribers are not responsible for the current state of the V24 flow. The credits for the data can be released only by the V24 Master subscriber.
The prototype of this API gives the definition of the API along with the parameters that need to be passed.

Prototype:

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

Parameters:

Handle: Handle returned by the adl_fcmSubscribe function.

NbCredits: This indicates the number of credits to release for this flow. If this number is greater than the number of previously received data blocks, all the credits are released. If an Open AT Application wants to release all the received credits, it should call the adl_fcmReleaseCredits API with NbCredits parameter set to 0xFF.

Returned Values:

OK is returned on successful subscription of the IO flow.

ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.

ADL_RET_ERR_BAD_HDL is returned if the handle is a V24 one (excluding the Master).

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

4.4. Switching Serial Link State

The adl_fcmSwitchV24State function switches the V24 serial link state to the AT mode or to the Data mode. The operation is effective only when the control event handler has received an ADL_FCM_EVENT_V24_XXX_MODE event. Only the V24 MASTER subscriber can use this API.
The prototype of this API gives the definition of the API along with the parameters that need to be passed.

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

Parameters:

Handle: Handle returned by the adl_fcmSubscribe function.

V24State: Serial link state to switch to. The allowed values are defined below:

```
ADL_FCM_V24_STATE_AT
ADL_FCM_V24_STATE_DATA
```

Returned Values:

OK is returned on successful subscription of the IO flow. The control handler will receive

ADL_FCM_EVENT_V24_XXX_MODE event when the serial link state has changed.

ADL_RET_ERR_PARAM is returned if one parameter has an incorrect value.

ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.

ADL_RET_ERR_BAD_HDL is returned if the handle is not the V24 MASTER one.

ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

4.5. Sending Data on the Requested FCM Flow

This function sends a data block on the requested flow.
The prototype of this API gives the definition of the API along with the parameters that need to be passed.

Prototype:

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

Parameters:

Handle: Handle returned by the adl_fcmSubscribe function.

Data: The buffer that must be written.

Returned Values:

OK is returned on successful subscription of the IO flow.

ADL_FCM_RET_OK_WAIT_RESUME is returned on success, (but the last credit was used).

ADL_RET_ERR_PARAM is returned is a parameter has an incorrect value.

ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.

ADL_RET_ERR_BAD_STATE is returned if the flow is not ready to send data.

ADL_FCM_RET_ERR_WAIT_RESUME is returned if the flow has no more credit to use.

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

ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.

NOTE :
*Unlike the standard Open AT OS interface, the Data block is not released by the adl_fcmSendData() API. The Open AT Application can use any u8 * buffer.*

4.6. Sending Unprocessed Data on the FCM FLOW

This function sends a data block on the requested flow. This API does not perform any processing on the provided data block, which is sent directly on the flow.
The prototype of this API gives the definition of the API along with the parameters that need to be passed.

Prototype:

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

Parameters:

Handle: Handle returned by the adl_fcmSubscribe function.

DataBlock: The data block buffer must be written using the following type:

```
typedef struct
{
u16 Reserved1 [4];
u32 Reserved3;
u16 Data Length;          /* Data length */
u16 Reserved2 [5];
u8 Data[1];               /* Data to send */
} adl_fcmDataBlock_t;
```

The block must be dynamically allocated and filled by the Open AT Application, before sending it to the function. The allocation size must 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 is returned on successful subscription of the IO flow.
ADL_FCM_RET_OK_WAIT_RESUME is returned on success, (but the last credit was used). On ADL_FCM_RET_OK_WAIT_RESUME returned value, the subscriber must wait for an ADL_FCM_EVENT_RESUME event on Control Handler to continue sending data.
ADL_RET_ERR_PARAM is returned if parameter has an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if the flow is not ready to send data.
ADL_FCM_RET_ERR_WAIT_RESUME is returned if the flow has no more credit to use. On ADL_FCM_RET_ERR_WAIT_RESUME returned value, the subscriber must wait for an ADL_FCM_EVENT_RESUME event on Control Handler to continue sending data.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.

NOTE :
As the standard Open AT OS interface, the data block will be released by the adl_fcmSendDataExt () API on OK and ADL_FCM_RET_OK_WAIT_RESUME return values. The Open AT Application must only use the dynamic allocated buffers.

4.7. Get Buffer State

This function gets the buffer status for requested flow handle.
The prototype of this API gives the definition of the API along with the parameters that need to be passed.

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 (to and from the Open AT Application), this parameter specifies the direction in which the buffer status is requested. The possible values are:

```
ADL_FCM_WAY_FROM_EMBEDDED,
ADL_FCM_WAY_TO_EMBEDDED
```

Returned Values:

ADL_FCM_RET_BUFFER_EMPTY is returned if the requested flow and way buffer is empty.

ADL_FCM_RET_BUFFER_NOT_EMPTY is returned if the requested flow and way buffer is not empty, and the FCM is still processing data on this flow.

ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.

ADL_RET_ERR_PARAM is returned if the way parameter value is out of range.

4.8. adl_fcmIsAvailable function

This API is used to check if a particular flow is available for FCM service or not.

Prototype:

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

Parameters:

Flow: The flow (UART, USB, GSM etc) to query for the availability. The value this parameter can take is defined by enum adl_fcmFlow_e.

Returned Values:

TRUE is returned if the specified port is available for FCM service.
FALSE is returned if the specified port is not available for FCM service.

5. Sample Code

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

/* Global variable */
u8 Handle;     //FCM handle
bool fcmCtrlH (adl_fcmEvent_e event );
bool fcmDataH (u16 DataLen, u8 * Data);
void adl_main (adl_InitType_e InitType )
{
  Handle=adl_fcmSubscribe(ADL_FCM_FLOW_V24_UART1,fcmCtrlH, fcmDataH);
  TRACE (( 1, "Embedded Application: Main" ));
}
bool fcmCtrlH ( adl_fcmEvent_e event )
{
  TRACE ((1, "Control event received --> %d", event));
  switch(event)
  {
    case ADL_FCM_EVENT_FLOW_OPENNED:
      TRACE (( 1, "Flow Opened" ));
      /* Switching V24 state from AT to Data. */
      adl_fcmSwitchV24State(Handle, ADL_FCM_V24_STATE_DATA);
      break;
    case ADL_FCM_EVENT_V24_DATA_MODE:
      TRACE (( 1, "Flow in Data Mode" ));
      /* sending data to the external application via V24 serial link */
      adl_fcmSendData(Handle, "This is a test case", 20);
      break;
  }
  return TRUE;
}

bool fcmDataH (u16 DataLen, u8 * Data)
{
  // The manipulation of the data to be done here
  adl_fcmSendData(Handle, Data, DataLen);
  return TRUE;
}
```

Summary

The following points have been covered in this chapter

- **The ADL library provides FCM APIs for the handling of the different Input Output flows (V24, GSM and GPRS).**
- **Subscribe to the flows using the adl_fcmSubscribe API.**
- **The V24 flow has both the Data and Command modes. In the Command mode, AT commands can be sent. Data can be sent in the Data mode.**
- **The adl_fcmSwitchState API is used to switch between the two states in the V24 mode.**
- **The data that comes from different flows is captured in the Data Handlers.**
- **The FCM related events are received in the control handlers.**

CHAPTER 17

Open UART

1. Objective

The ADL Open Device service provides a raw access to any device behaving as a serial port. In order to get a raw access to the device, software block component called Service Provider (SP) supplies APIs which allows manipulating the device. The 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.

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 the device behaving via SPs defined by Firmware or Open AT application.
- Services provided 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).

The diagram below illustrates a typical mechanism between SU and SP:

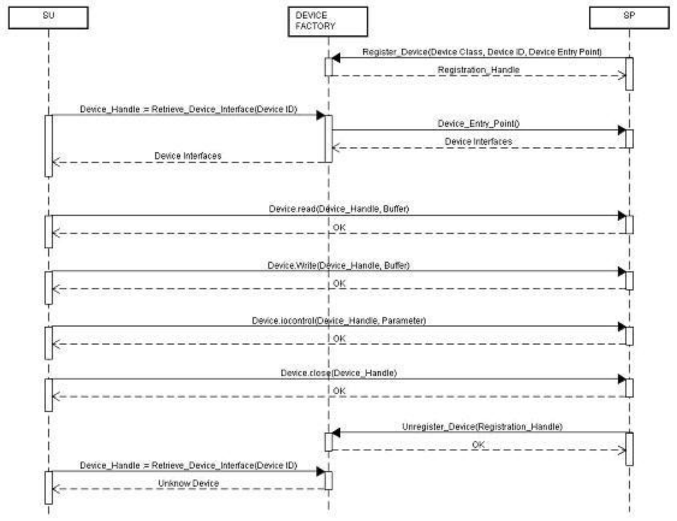
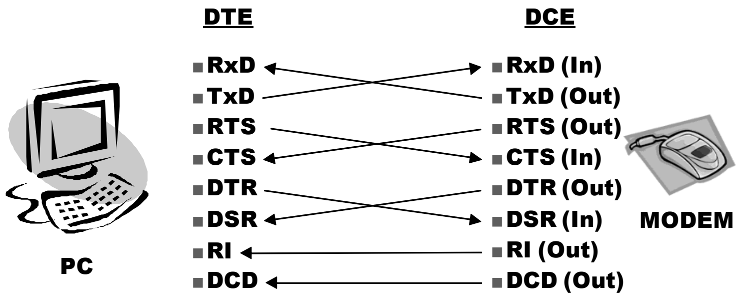

SU
DEVICE FACTORY
SP
Register_Device(Device Class, Device ID, Device Entry Point)
Registration_Handle
Device_Handle = Retrieve_Device_Interface(Device ID)
Device_Entry_Point()
Device Interfaces
Device Interfaces
Device.read(Device_Handle, Buffer)
OK
Device.Write(Device_Handle, Buffer)
OK
Device.iocontrol(Device_Handle, Parameter)
OK
Device.close(Device_Handle)
OK
Unregister_Device(Registration_Handle)
OK
Device_Handle = Retrieve_Device_Interface(Device ID)
Unknow Device

Figure 59: Illustration of the mechanism between the Service Provider and the Service User

2. Understanding UART Signals

The UART signals (RTS, CTS, DTR, DSR, DCD and RI) are briefly described as given below.



RxD / TxD lines are minimal signals required to exchange Data.

If data transmissions are too fast compared to the data processing, overruns (lost of chars, hard/soft FIFO overrun) could occur.

Best results are achieved if flow control is activated to avoid overruns. (UART FIFO overruns if remote sends too many data and cannot be processed fast enough, therefore flow control is needed to stop / resume data flow).

Soft flow control XON/XOFF characters can be used if only RxD/TxD line are available without RTS/CTS.

2.1. RTS (Request To Send)

Used by DCE (Data Communication Equipment) to indicate that modem is able to receive data.

At start up, DTE (Data Terminal Equipment) application see the modem CTS high, meaning that DTE can send data to DCE.

2.2. CTS (Clear To Send)

Used by DCE to determine if DTE (eg. PC) is able to receive.
At start-up, if CTS is not asserted, it is considered as asserted in order to prevent “CTS less” application from being stuck.

2.3. DTR (Data Terminal Ready)

Used by DCE to indicate whether it is online or not (unless deactivated by an AT command).
DTE application DSR is high when modem is online (DATA transmission mode) and low if modem is in offline mode (AT mode).

2.4. DSR (Data Set Ready)

Used by DTE to make DCE hang up during a DATA transmission.
Or used to enter in Sleep Mode (if activated in Modem).

2.5. RI (Ring Indicator)

Used by DCE to indicate an incoming event (SMS, Call, IP packet)
When an incoming call arrives, the RI triggers periodically.

2.6. DCD (Data Carrier Detect)

Used by DCE to depict if Modem has a connection established (different from DTR which depicts online mode).

Modem can be either in connected mode or not connected mode. A modem is connected if a GSM data call or GPRS data session is running. Once connected, a modem is Online and all the data from DTE to DCE are considered as data payload and sent to the remote modem. A modem is Offline if Data sent to or received from the DTE are AT commands.

3. Open UART Interface

ADL provides Open UART Interface to give direct access to the embedded module UART Service Providers. UART Service Provider is required to implement the Open UART Interface. 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 Firmware. Services provided by the UART service providers can be accessed by the Firmware or the Open AT application but not both simultaneously.

To be accessible, a SP of device has to be previously registered in the Firmware. Then its services can be accessed by the SU.

Each SP provides 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.

4. Open UART 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

There are five request functions that are provided by the Open UART interface:

- **An open function to:**

  Optionally retrieve the UART service provider's interface (through a generic interface container).

  Retrieve a unique UART service provider reference (handle) which shall be subsequently provided as parameter to the rest of the request functions.

  Optionally install event handlers to manage the UART service provider notifications.

  Set the role of the UART which allows the service user to indicate to the UART service provider which running mode shall be established.

  Optionally set the line coding parameters.

  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).

There are six notifications that the UART service provider notifies the event occurances to the UART service user:

- The completion of the current emission (write completion at the byte level and also 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 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. Open Function

There is no 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.

Prototype:

```
s16 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.

Sample code

```
#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 );
 …
}
```

4.2. Read Function

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.

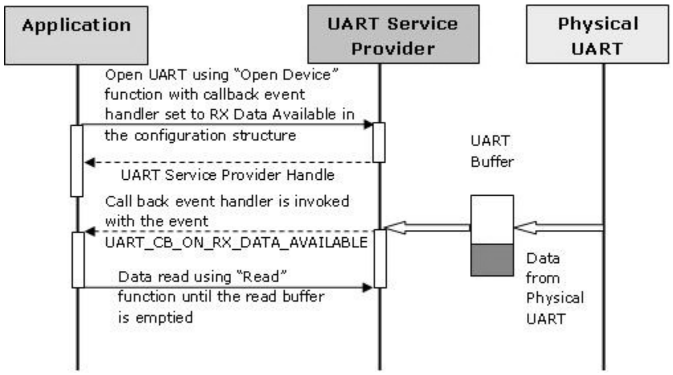

Application
UART Service Provider
Physical UART
Open UART using "Open Device" function with callback event handler set to RX Data Available in the configuration structure
UART Service Provider Handle
UART Buffer
Call back event handler is invoked with the event UART_CB_ON_RX_DATA_AVAILABLE
Data read using "Read" function until the read buffer is emptied
Data from Physical UART

Figure 60: Synchronous Read

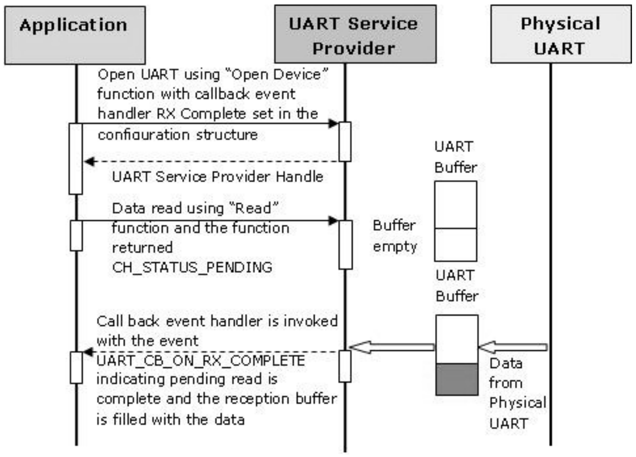

Application
UART Service Provider
Physical UART
Open UART using "Open Device" function with callback event handler RX Complete set in the configuration structure
UART Service Provider Handle
UART Buffer
Data read using "Read" function and the function returned CH_STATUS_PENDING
Buffer empty
UART Buffer
Call back event handler is invoked with the event UART_CB_ON_RX_COMPLETE indicating pending read is complete and the reception buffer is filled with the data
Data from Physical UART

Figure 61: Asynchronous Read

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_Open Device 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.

Sample Code

```
#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 sample code section 17.3.2 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.read( 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;
  }
}
```

4.3. 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 synchronous (S) mode.

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. Application has then the possibility to provide further data to write, directly in the On TX Complete/Empty event handler, without any further call to the Write API.

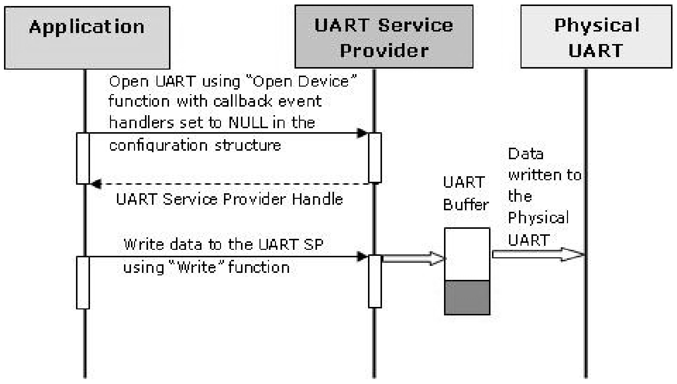

Application
UART Service Provider
Physical UART
Open UART using "Open Device" function with callback event handlers set to NULL in the configuration structure
UART Service Provider Handle
Write data to the UART SP using "Write" function
UART Buffer
Data written to the Physical UART

Figure 62: Synchronous Write

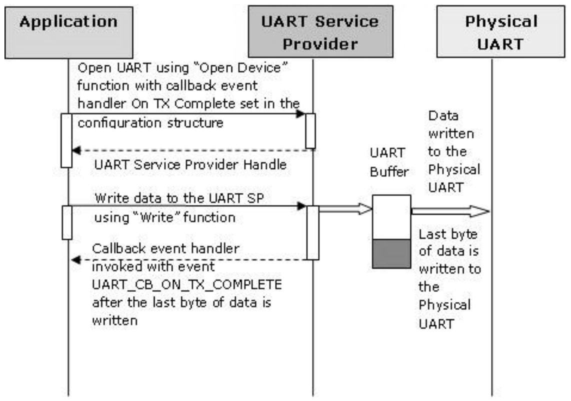

Application
UART Service Provider
Physical UART
Open UART using "Open Device" function with callback event handler On TX Complete set in the configuration structure
UART Service Provider Handle
Write data to the UART SP using "Write" function
Callback event handler invoked with event UART_CB_ON_TX_COMPLETE after the last byte of data is written
UART Buffer
Data written to the Physical UART
Last byte of data is written to the Physical UART

Figure 63: Asynchronous Write – Byte Level Notification

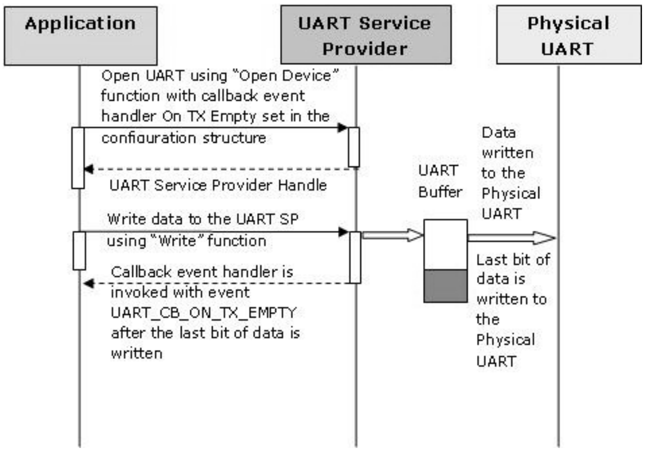

Application
UART Service Provider
Physical UART
Open UART using "Open Device" function with callback event handler On TX Empty set in the configuration structure
UART Service Provider Handle
Write data to the UART SP using "Write" function
Callback event handler is invoked with event UART_CB_ON_TX_EMPTY after the last bit of data is written
UART Buffer
Data written to the Physical UART
Last bit of data is written to the Physical UART

Figure 64: Asynchronous Write – Bit Level Notification

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.

4.4. 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.

pParam: IO command parameter. Type of this parameter depends on the Cmd parameter value.

Returned values:

Depend on the IO command type.

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.

4.6. 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.

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.

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 to a NULL value.

Returned values:

Not Applicable.

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.

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 Signal State Change 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.

Returned values:

Not Applicable.

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 Error 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.

Returned values:

Not Applicable.

NOTE :
Please refer to the ADL_User_Guide for more details on the structures definition referred in the above functions

5. Benefits of Open UART

- Expands control of external chipsets through directly accessing UART.
- Expands the range of external devices the embedded Module can access.
- Can achieve real time durations.
- Low level interrupts response time guaranteed on Read/Write operation.
- Faster & Flexible compared to Flow Control Manager.
- Can overcome the overhead caused due to the processing of information through the various layers (such as ADL) between the Open AT application and the low level UART in case of FCM service.
- Used for metering applications, UART Direct Access allows handling Zigbee or PLC chips.

6. Overhead of Open UART over FCM

Overhead to manage the flow control, UART settings etc (only if variable UART_FC_RTS_CTS in the structure sUartFlowCtrl_t is not set) from the application side compared to FCM service in which case is handled by the ADL layer.

NOTE :
FCM and Open UART cannot be simultaneously used on the same port.

Summary

The following points have been covered in this chapter:

- **ADL Open Device service provides a raw access to any device behaving as a serial port. In order to get a raw access to the device, software block component called Service Provider (SP) supplies APIs which allows manipulating the device.**
- **ADL provides Open UART Interface to give direct access to the embedded module UART Service Providers (SP).**
- **Open UART Service Provider provides five request functions i.e. read, write, I/O control and close functions and provides six notifications which include write completion at the byte level, write completion at the bit level, availability of received data, change in the signal states, errors and completion of the current reception.**

CHAPTER 18

General Purpose Input Output

1. Objective

This chapter introduces the GPIO (General Purpose Input Output) APIs available in the ADL library. It describes the General Purpose Input Output pins, their usage, and the procedure to write into and read from the GPIOs.

2. General Purpose Input Output

GPIOs are the pins that can be used to send and receive data. Hence when the embedded module is connected to any other device the exchange of data can be done with the help of GPIOs. The data over the GPIO pin is sent bit-by-bit. The GPIO pins can be of the following types:

- GPO (General purpose Output): This pin can be used to send the data (output only).
- GPI (General purpose Input): This pin can be used to receive the data (input only).
- GPIO (General purpose Input Output): This pin can be used to send and receive data (input and output)

These pins can be used when the embedded module needs to send data to trigger external devices, or if the embedded module needs to be triggered by an external device.

3. Availability of GPIOs

3.1. GPIOs in AirPrime Q24NG embedded module.

AirPrime Q24NG series of embedded module provides the following GPIO/GPI/GPO:

- 6 GPIOs
- 4 GPOs
- 1 GPI's

3.2. GPIOs in AirPrime Q26XX embedded module.

AirPrime Q26XX series of embedded module provides 44 GPIOs. Out of these 36 GPIOs are multiplexed with features such as bus, keyboard, UART etc.

4. GPIO Service Provided by ADL Library

The number of available GPIOs differs for each embedded module. This depends on the GPIOs that are used by the lower layer for the embedded module. The GPIO IDs that are used to subscribe to the GPIOs depend on the embedded module type that is used. Please use adl_ioGetCapabilities() API to get information about the GPIOs available for different embedded modules.

5. GPIO Service Related Structures and Enumerations

This section lists the content of structures and enumerations used by GPIO service.

5.1. 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;
```

5.2. 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 adl_ioLabel_e type
- b16-b31 usage depends of the command
- see adl_ioLevel_e type
- see adl_ioDir_e type
- see adl_ioStatus_e type
- see adl_ioCap_e type
- see adl_ioError_e type

5.3. 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. Note that b14-b15 is reserved.

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

5.4. The adl_ioLevel_e Type

This type lists the level field definition (b16 of adl_ioDefs_t).
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

5.5. The adl_ioDir_e Type

This type lists the direction field definition (b17-b18 of adl_ioDefs_t).

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;
```

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

5.6. The adl_ioError_e Type

This type lists the direction field definition (b28-b31 of adl_ioDefs_t).

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;
```

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)

5.7. The adl_ioCap_e Type

This type lists the direction field definition (b21-b22 of adl_ioDefs_t).

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;
```

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

5.8. The adl_ioStatus_e Type

This type lists the direction field definition (b19-b20 of adl_ioDefs_t).

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;
```

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

5.9. 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;
```

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.

6. GPIO APIs

The GPIO APIs provided by the ADL library, are used to control the input-output pins. These pins can be subscribed using the adl_ioSubscribe API. The subscribed input-output pins can be used to send and receive the data using the APIs provided in the ADL library.

6.1. Get List of GPIO Capabilities

This API is used to get the embedded module GPIO capabilities list. For each hardware available GPIO, the embedded module shall add an item in the GPIO capabilities list.

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 capabilities information (using adl_ioDefs_t ** type).

GpioTypeNb: Returns the number of each GPIO, GPO and GPI. This is an optional parameter. This parameter will not be used if it is set to NULL.

Returned Values:

OK is returned on success.

ADL_RET_ERR_PARAM is returned if one of the parameter is incorrect.

6.2. Subscribing to GPIO Related Events

This API is used to subscribe to GPIO related events. A call back function is also setup which will receive the GPIO related events**.** The GPIO service will be available for subscription only after the subscription of the GPIO related events.

Prototype:

s32 adl_ioEventSubscribe(adl_ioHdlr_f GPIOEventHandler)

Parameters:

GPIOEventHandler: Call-back function used to report all the events related to GPIO.
Prototype:
```
typedef void (*adl_ioHdlr_f) ( s32 GpioHandle, adl_ ioEvent_e   Event,
                               u32 Size, void *param)
```
Parameters:

GPIOHandle: GPIO subscription handle.
Size: Number of item in param table.
Event: Event are received as event identifier.
Param: This parameter is dependent on the type of event.

ADL_IO_EVENT_INPUT_CHANGED: One or several of the subscribed input have changed. Read the value table.
Outputs available for each array element:

the GPIO label (Refer adl_ioLabel_e).

the GPIO level (Refer adl_ioLevel_e).
the GPIO error information (Refer adl_ioError_e).

Returned Values:

A positive or null handle is returned on successful subscription. This handle will be used for the corresponding GPIO related operation.
ADL_RET_ERR_PARAM is returned if the parameter is invalid.
ADL_RET_ERR_NO_MORE_HANDLE is returned if GPIO service is subscribed for more than 128 times.
ADL_RET_ERR_SERVICE_LOCKED is returned if this API is called from a low level interrupt handler.

6.3. Unsubscribing from GPIO Events

This API is available only from Open AT OS v4.1x and is used to unsubscribe from the GPIO event notification.

Prototype:

s32 adl_ioEventUnsubscribe (s32 GpioEventHandle)

Parameters:

GPIOEventHandle: Handle returned by the subscription function.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if GPIO service is unsubscribed with wrong handle.
ADL_RET_ERR_NOT_SUBSCRIBED is returned if no GPIO service is subscribed.
ADL_RET_ERR_BAD_STATE is returned if a polling process is running in the current handle.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

6.4. Subscribe to GPIO Service

The adl_ioSubscribe () API is used to subscribe to the GPIO functionality. The user can mention the GPI's, GPO's and the GPIO's that are to be used.
The API will allow the polling for the GPI pins and the input pins can be read at any time after the subscription. Similarly the output pins can be set any time after the subscription.

Prototype:

s32 adl_ioSubscribe (u32 GpioNb, adl_ioDefs_t * GpioConfig, u8 PollingTimerType, u32 PollingTime, s32 GpioEventhandle)

Parameters:

GPIONb: Size of GpioConfig array.

GpioConfig: GPIO subscription configuration array. The number of elements in this array is "GpioNb". For each element, the adl_ioDefs_t structure members have to be configured.

Inputs to set for each array element:

Label of the GPIO to subscribe (Refer adl_ioLabel_e)

GPIO direction (Refer adl_ioDir_e)
GPIO level, only if the GPIO is an output (Refer adl_ioLevel_e)

Outputs available for each array element:
GPIO error information (Refer adl_ioError_e)

PollingTimerType: Type of timer using which the polling will be performed. For instance, the value will be ADL_TMR_TYPE_100MS for a timer of 100 millisecond granularity. The value should be ADL_TMR_TYPE_TICK for a timer of 18.5 millisecond granularity.

PollingTime: The number of units (specified in PollingTimerType) after which the polling will be performed.
GpioEventHandle: GPIOEventHandle returned by adl_ioEventSubscribe () function. If no polling is requested, this parameter is ignored.

Returned Values:
A positive or null handle is returned on success.
ADL_RET_ERR_PARAM is returned if a parameter has an incorrect value.
ADL_RET_ERR_DONE refers to the field adl_ioError_e for more information.
ADL_RET_ERR_NO_MORE_TIMERS is returned if no more timers are free to perform polling.
ADL_RET_ERR_NO_MORE_HANDLES is returned if no more GPIO handles are available.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level Interrupt handler.

6.5. Unsubscribing from GPIO Service

All the GPIOs subscribed earlier by the adl_ioSubscribe function, are unsubscribed by the adl_ioUnsubscribe API. The handle returned by earlier adl_ioSubscribe API must be provided to this function:

Prototype:
```
s32 adl_ioUnsubscribe (s32 Handle)
```

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

Returned Values:
OK is returned on success of execution of the function.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function isalled from a low level Interrupt handler.

6.6. Set Direction (Input/Output) Of GPIO

This API is used to modify the direction (input or output) of the previously allocated GPIOs.

Prototype:
```
s32 adl_ioSetDirection (s32 GpioHandle, u32 GpioNb, adl_ioDefs_t *GpioDir)
```

Parameters:
GpioHandle: Handle returned by the subscription function.

GpioNb: Size of GpioRead array.

GpioDir: GPIO direction. It is defined as

Inputs to set for each array element:
the label of the GPIO to modify (Refer adl_ioLabel_e)
the new GPIO direction (Refer adl_ioDir_e)

Outputs available for each array element:
the GPIO error information (Refer adl_ioError_e)

Returned Values:

OK is returned on success of execution of the function.
ADL_RET_ERR_PARAM is returned 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 is returned if the handle is unknown.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level Interrupt handler.

6.7. Reading from Multiple GPIOs

The adl_ioRead () function allows to read from subscribed GPIOs.

Prototype:

```
s32 adl_ioRead ( s32 GpioHandle, u32 GpioNb, adl_ioDefs_t* GpioRead)
```

Parameters:

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

GpioNb: Size of GpioRead array.

GpioRead: The structure which identifies the GPIO to be read.

Inputs to set for each array element:

The label of the GPIO to read (Refer adl_ioLabel_e).

Outputs available for each array element:

The GPIO level value (Refer adl_ioLevel_e).

The GPIO error information (Refer adl_ioError_e)

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if a parameter is invalid.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.

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.

6.8. Reading from Single GPIO

This API is used to read single GPIO from the previously allocated handle.

Prototype:

s32 adl_ioReadSingle(s32 GpioHandle, adl_ioDefs_t Gpio)

Parameters:

GpioHandle: Handle returned by the subscription function.

Gpio: GPIO identifier defined in the union adl_ioLabel_e.

Returned Values:

GPIO read value is returned on success (1 for a high level or 0 for a low level).
ADL_RET_ERR_UNKNOWN_HDL is returned if correct handle is specified.
ADL_RET_ERR_PARAM is returned if the parameters are incorrect.
ADL_RET_ERR_BAD_STATE is returned if the GPIO is not subscribed as output.

6.9. Writing on Multiple GPIOs

The adl_ioWrite () function allows to write into subscribed GPIOs.

Prototype:

s32 adl_ioWrite (s32 GpioHandle, u32 GpioNb, adl_ioDefs_t* GpioWrite)

Parameters:

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

GpioNb: Size of GpioRead array.

GpioWrite: The structure which identifies the GPIO to be written

Inputs to set for each array element:

the label of the GPIO to write (Refer adl_ioLabel_e)
the new GPIO level (Refer adl_ioLevel_e)

Outputs available for each array element:

the GPIO error information (Refer adl_ioError_e)

Returned Values:

OK is returned on success.

ADL_RET_ERR_PARAM is returned if a parameter is invalid.
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 is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if one of the GPIO is not subscribed as output (for which a write is requested).

6.10. Writing on Single GPIO

This API is used to write a value to the GPIO from the previously allocated handle.

Prototype:

s32 adl_ioWriteSingle(s32 GpioHandle, adl_ioDefs_t Gpio, bool State)

Parameters:

GpioHandle: Handle returned by the subscription function**.**

Gpio: Identifier of GPIO value of the identifier. Refer to adl_ioLabel_e for more details.

State: Value to be set on output.

TRUE for a high level
FALSE for a low level

Returned Values:

OK: is returned on successful write operation.
ADL_RET_ERR_UNKNOWN_HDL is returned if incorrect handle is specified.
ADL_RET_ERR_PARAM is returned if the parameters are incorrect.
ADL_RET_ERR_BAD_STATE is returned if the GPIO is not subscribed as input.

7. Sample Code

```
#include "adl_global.h"
#include "adl_gpio.h"

const u16 wm_apmCustomStackSize = 4096;

// Subscription data
#define GPIO_COUNT1 2
#define GPIO_COUNT2 1
const u32 My_Gpio_Label1 [ GPIO_COUNT1 ] = { 1 , 2 }
const u32 My_Gpio_Label2 [ GPIO_COUNT2 ] = { 3 }
const adl_ioDefs_t MyGpioConfig1 [ GPIO_COUNT1 ] =
{ { ADL_IO_GPIO | My_Gpio_Label1 [ 0 ] | ADL_IO_DIR_OUT | ADL_IO_LEV_LOW },
{ ADL_IO_GPIO | My_Gpio_Label1 [ 1 ] | ADL_IO_DIR_IN } };
const adl_ioDefs_t MyGpioConfig2 [ GPIO_COUNT2 ] =
{ { ADL_IO_GPIO | My_Gpio_Label2 [ 0 ] | 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 & Param[ My_Loop ] ) && ADL_IO_GPO )
    {
      TRACE (( 1, "GPO %d new value: %d",
       ( Param[ My_Loop ] ) & ADL_IO_NUM_MSK ,\
       ( Param[ My_Loop ]) & ADL_IO_LEV_MSK ) && \
        ADL_IO_LEV_HIGH ));
    }
    else
    {
      TRACE (( 1, "GPIO %d new value: %d", \
      ( Param[ My_Loop ] ) & ADL_IO_NUM_MSK , \
      ( Param[ My_Loop ] ) & ADL_IO_LEV_MSK ) && \
      ADL_IO_LEV_HIGH ));
    }
   }
  }
  break;
 }
}

void MyFunction ( void )
{
 s32 ReadValue;
 // Subscribe to the GPIO event service
 MyGpioEventHandle = adl_ioEventSubscribe ( MyGpioEventHandler );
 // Subscribe to the GPIO service (One handle without polling,
 // one with a 100ms polling process)
 MyGpioHandle1 = adl_ioSubscribe ( GPIO_COUNT1,MyGpioConfig1,0,0,0 );
 MyGpioHandle2 = adl_ioSubscribe ( GPIO_COUNT2, MyGpioConfig2, \
              ADL_TMR_TYPE_100MS, 1, MyGpioEventHandle );
 // Set output
 adl_ioWriteSingle ( MyGpioHandle1, ADL_IO_GPIO | My_Gpio_Label1 \
                [ 0 ] , TRUE );
 // Read inputs
 ReadValue = adl_ioReadSingle (MyGpioHandle1, ADL_IO_GPIO |
 My_Gpio_Label1 [ 1 ] );
 ReadValue = adl_ioReadSingle (MyGpioHandle2, ADL_IO_GPIO |
 My_Gpio_Label2 [ 0 ] );
 // Unsubscribe from the GPIO services
 adl_ioUnsubscribe ( MyGpioHandle1 );
 adl_ioUnsubscribe ( MyGpioHandle2 );
 // Unsubscribe from the GPIO event service
 adl_ioEventUnsubscribe ( MyGpioEventHandle );
}

void adl_main (adl_InitType_e adlInitType)
{
 TRACE((1,"In main function"));
 MyFunction();
}
```

Summary

The following points have been covered in this chapter

- **The ADL library provides GPIO APIs to handle the various input-output pins**
- **Using the adl_ioSubscribe API to subscribe to GPIOs.**
- **The GPIOs can be GPIs or GPOs or GPIOs.**
- **To unsubscribe from a GPIO, use the adl_ioUnSubscribe API. The handler that is returned by the adl_ioSubscribe API must be given as parameter to adl_ioUnSubscribe API.**
- **The APIs adl_ioWrite and adl_ioRead are used to read and write data into the subscribed GPIOs.**
- **The APIs adl_ioSingleWrite and adl_ioSingleRead are used to write/read from a single GPIOs.**

CHAPTER 19

Bus Service

1. Objective

This chapter describes the bus service and its usage in the embedded module. ADL provides the bus service to handle two SPI buses, an I2C bus and a parallel bus too.

2. Bus Description

Bus refers to a common data path or channel between multiple devices acting as peripherals. The following sections give a brief explanation of the three kinds of buses supported in embedded module.

2.1. SPI Bus

SPI (Serial Peripheral Interface) bus is a simple asynchronous serial interface for connecting low-speed, external devices using a minimal number of wires. The devices connected to the SPI bus may be classified as Master or Slave devices. A Master device initiates an information transfer on the bus and generates clock and control signals. A Slave device is controlled by the master through a slave select (chip enable) line and is active only when selected. A device can possess the functionality of a master and a slave; however, only one master can control the bus. If a single slave is used – the SPI bus is a four-wire protocol. In the case of multiple slaves, there are (3+n) wires, where n is the number of slaves.

The signals pins available on an SPI bus are:

- SCK (Serial Clock): serial clock signal to regulate the flow of data bits. It is generated by the master to synchronize data transfer.

- MOSI (Master Out Slave Input): used to transfer data from the master to the slave device.

- MISO (Master In Slave Out): used to transfer data from the slave to the master device.

- SS (Slave Select): used to select a particular slave device when multiple slave devices are connected to a master.

Additionally the Load and Busy signals can be used to synchronise the data transfer. For more information see the ADL user guide.

When using SPI, data is simultaneously transmitted and received, making it a full-duplex protocol.
The schematic diagram and timing diagrams of unidirectional and bidirectional SPI buses are shown below:

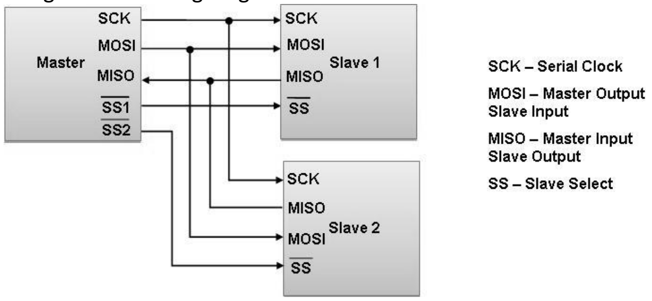


Figure 65: Schematic diagram of Unidirectional SPI bus

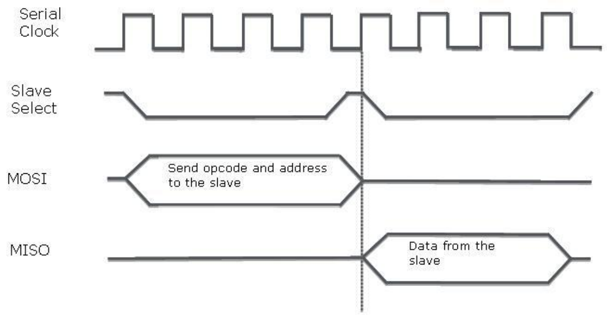


Figure 66: Timing diagram for Unidirectional SPI bus with flash device as example

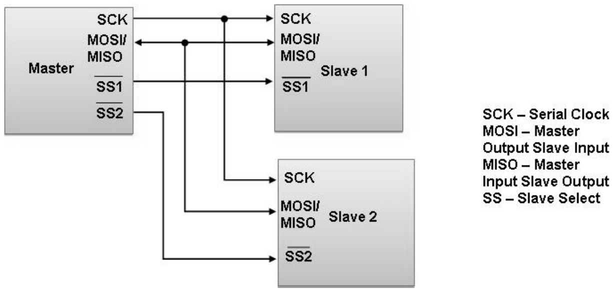


Figure 67: Schematic diagram of Bidirectional SPI bus

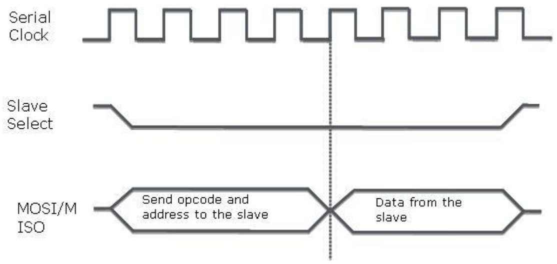


Figure 68: Timing diagram for Bi-directional SPI bus with flash device as example

2.2. I2C Bus

I2C is a two-wire serial bus protocol to carry information between the two peripheral devices. The bus consists of two signals – serial data (SDA) and serial clock (SCL), both being bidirectional lines. When the bus is free, both the lines are HIGH.

Each device connected by the I2C bus is identified by a unique address. In case of a data transfer, each device can be classified as a Master or a Slave. Here too, a Master is a device which initiates the data transfer and generates the clock signal to permit that transfer.

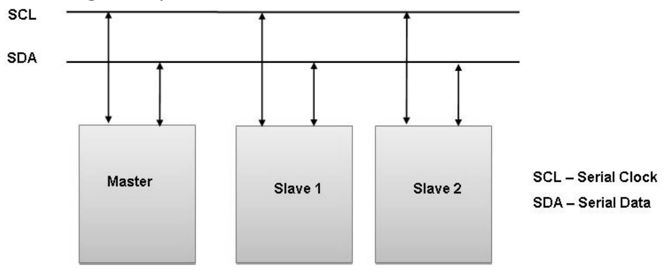


Figure 69: Schematic diagram of typical I2C bus with master and two independent slaves

2.3. Parallel Bus

Parallel bus can be used for device which supports high speed data transfer. The parallel bus data width can be up to 32 bits thus allowing transfer of 32 bits in parallel.

3. Bus APIs

This section describes the APIs that are available in embedded module for bus management. Include the adl_bus.h header file to use the bus service.

3.1. Subscribing to Bus Service

This following API is used to subscribe to a bus service and manage bus related events. Simultaneously 8 configurations can be configured and can be connected to 8 different devices.

Prototype:

```
s32 adl_busSubscribe (adl_busID_e BusId, u32 BlockId, void * BusParam )
```

Parameter:

BusType: The bus which should be subscribed. The bus type is defined to the enum adl_busType_e.

```
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;
```

BlockId: Id of the block to use in the range 1-N, where N (i2c_NbBlocks/ spi_NbBlocks/Para_NbBlocks) is specific to each bus type and embedded module platform.

BusParam: Subscribed bus configuration parameters:

SPI: The setting for the SPI bus. This structure adl_busSPISetting_t is defined below:

```
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;
```

Clk_Speed: This is the speed of the clock of the SPI bus.

Clk_Mode: There are four modes that are supported.
- ADL_BUS_SPI_CLK_0: rest state 0, Data valid on rising edge.
- ADL_BUS_SPI_CLK_1: rest state 0, data valid on falling edge.
- ADL_BUS_SPI_CLK_2: rest state 1, data valid on rising edge.
- ADL_BUS_SPI_CLK_3: rest state 1, data valid on falling edge.
- ADL_BUS_SPI_CLK_MODE_MICROWIRE: Microwire mode

ChipSelect: Pin to use for chip select signal. Defined values are:

- ADL_BUS_SPI_ADDR_CS_GPIO: Use GPIO as chip select signal.
- ADL_BUS_SPI_ADDR_CS_HARD: Use reserved hardware chip select pin.
- ADL_BUS_SPI_ADDR_CS_NONE: Chip select is not handled by the bus service.

ChipSelectPolarity: Polarity of the chip select signal, defined values are:
- ADL _BUS_SPI_CS_POL_LOW: Chip select is active in low state.
- ADL _BUS_SPI_CS_POL_HIGH: Chip select is active in low state.

LsbFirst: Priority of the data transfer through the SPI bus, various defined values are:

- ADL_BUS_SPI_MSB_FIRST: Data is sent with MSB first.
- ADL_BUS_SPI_LSB_FIRST: Data is sent with LSB first.

GpioChipSelect: It is used in case GPIO is chosen for the Chip select signal. It defines the label to be used for chip select signal.

LoadSignal: It defines the LOAD signal behaviour.
- ADL_BUS_SPI_LOAD_UNUSED: The Load signal is not used.
- ADL_BUS_SPI_LOAD_USED: The LOAD signal is used. The LOAD signal state changes on each written or read word; word size is defined by ADL_BUS_CMD_SET_DATA_SIZE IOCtl command. Please refer to the PTS document for more information about the LOAD signal.

DataLinesConf: It is used to define if the single line handles both the input and output. Following are the value defined:
- ADL_BUS_SPI_DATA_BIDIR: One PIN to handle both input and output.
- ADL_BUS_SPI_DATA_UNDIR: Two separate PIN for input and output.

MasterMode: It is used to define the SPI bus mode.
- ADL_BUS_SPI_MASTER_MODE: The SPI bus running in Master mode.
- ADL_BUS_SPI_SLAVE_MODE: The SPI bus running in Slave mode.

BusySignal: It used to define the LOAD signal behaviour.
- ADL_BUS_SPI_BUSY_UNUSED: The BUSY signal is not used. This is the default value when adl_busSubscribe function is used.
- ADL_BUS_SPI_BUSY_USED: The BUSY signal is used.

I2C: The setting for the I2C bus. The settings structure adl_busI2Csetting_t is given below

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

ChipAddress: It is used to set the N bit address on the I2C bus. Note that only b1 to bN bits are used to set the address. The b0 bit and most significant bytes are neglected.
N Value depends on the embedded module capabilities, and on the adl_busI2CSettings_t::AddrLength field configuration. For example, if the I2C address is set to A0, the ChipAddress parameter has to be set to the 0xA0 value.

Clk_Speed: Used to set the speed for the I2C bus. Following are the values defined.
- 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(3.4Mbits/s)

AddrLength: This parameter is used to set the I2C chip address length.

- ADL_BUS_I2C_ADDR_7_BITS: Chip address is 7 bits long. This is the default value if adl_busSubscribe function is used.
- ADL_BUS_I2C_ADDR_10_BITS: Chip address is 10 bits long.

MasterMode: It is used to define the I2C bus mode.

- ADL_BUS_I2C_MASTER_MODE: The I2C bus running in master mode. By default this is the mode which is used when I2C is subscribed using adl_busSubscribe function.
- ADL_BUS_I2C_SLAVE_MODE: The I2C bus running in Slave mode.

PARALLEL BUS: The setting for the parallel bus. The settings structure adl_busParallelsetting_t is given below .

```
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_busParallelsetting_t  ;
```

Width: Defines the width of the bus.

- 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

Mode: Define the standard parallel bus mode to be used.

- ADL_BUS_PARALLEL_MODE_ASYNC_INTEL: Standard Intel mode
- ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_HIGH: Standard Motorola mode with E signal high polarity
- ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_LOW: Standard Motorola mode with E signal low polarity
- ADL_BUS_PARALLEL_MODE_ASYNC_PAGE: Page mode
- ADL_BUS_PARALLEL_MODE_SYNC_READ_ASYNC_WRITE: Synchronous only in read operation
- ADL_BUS_PARALLEL_MODE_SYNC_READ_WRITE: Synchronous in both read and write operation

ReadCfg: Defines the timing configuration of the read process. This is defined by:

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

AccessTime: Access time. Refer Mode parameter and PTS for more information.

SetupTime: Setup time Refer Mode parameter and PTS for more information.

HoldTime: Hold time. Refer Mode parameter and PTS for more information.

TurnaroundTime: Turnaround time. Refer Mode parameter and PTS for more information.

OptoOpTurnaroundTime: Read-to-Read/Write-to-Write turnaround time. Refer Mode parameter and PTS for more information.

WriteCfg: Defines the timing configuration of the write process. This is defined by adl_busPrallelTimingCfg_t as mentioned for ReadCfg parameter.

Cs: Defines the chip select configuration of the parallel bus.

```
typedef struct
{
u8 Type;
u8 Id;
u8 Pad[2];
} adl_busParallelCs_t;
```

Type: Defines the chip select signal type

ID: Defines the chip select identifier used

PageCfg: Defines the configuration parameter for the page mode.

```
typedef struct
{
 u8 PageSize;
 u8 PageAccessCycles;
} adl_busParallelPageCfg_t;;
```

PageSize: Defines the page size

PageAccessCycles: Defines the access cycle between address change and valid data output.

SynchronousCfg: Defines the configuration parameter for the synchronous mode.

```
typedef struct
{
  u8 BurstSize;
  u8 ClockDivisor;
  s32 UseWaitEnable:1;
  s32 WaitActiveDuringWS:1;
  s32 Reserved:30;
} adl_busParallelSynchronousCfg_t;
```

BurstSize: Defines the burst size

ClockDivisor: Defines the main memory clock divider

UseWaitEnable:1: Defines the WS generation using WAIT#

WaitActiveDuringWS:1: Defines the WAIT# during or 1-cycle before WS

AddressPin: Selects the pin used for the parallel bus.

Returned values:

A positive or null handle is returned on success.
ADL_RET_ERR_PARAM is returned if parameter has an incorrect value.
ADL_RET_ERR_ALREAY_SUBSCRIBED is returned if required bus is already subscribed.
ADL_RET_ERR_BAD_HDL is returned if GPIO required for the bus configuration is currently unavailable.
ADL_RET_ERR_NOT_SUPPORTED is returned: if the required bus type is not supported by the embedded module on which the application is running.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.

3.2. Unsubscribing from the Bus Service

The following API should be used to unsubscribe from the bus service:

Prototype:

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

Parameters:

Handle: Handle returned by the adl_busSubscribe function.

Returned Values:

OK is returned on successfully unsubscribing from the bus service.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler

3.3. Modifying the Configuration of the Subscribed Bus

The following API should be used to modify the configuration and the behavior of the subscribed bus.

Prototype:

s32 adl_busIOCtl (u32 Handle, adl_busIoCtlCmd_e Cmd, void * Param)

Parameters:

Handle: Handle returned by the adl_busSubscribe function.

Cmd: Command to be executed.

- ADL_BUS_CMD_SET_DATA_SIZE (Available for the SPI bus only): Sets the size in bits of one data element.
- ADL_BUS_CMD_GET_DATA_SIZE (Available for the SPI bus only): Gets the size in bits of one data element.
- ADL_BUS_CMD_SET_ADD_SIZE (Available for the SPI and I2C bus only): Sets the size in bits of the address.
- ADL_BUS_CMD_GET_ADD_SIZE (Available for the SPI and I2C bus only): Gets the size in bits of the address.
- ADL_BUS_CMD_SET_OP_SIZE (Available for the SPI bus only): Sets the size in bits of the Opcode.
- ADL_BUS_CMD_GET_OP_SIZE (Available for the SPI bus only): Gets the size in bits of the Opcode.
- ADL_BUS_CMD_CLOCK (Available for the SPI and I2C bus only): Locks 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 used this block.
- ADL_BUS_CMD_UNLOCK (Available for the SPI and I2C bus only): Unlocks a bus.
- ADL_BUS_CMD_GET_LAST_ASYNC_RESULT (Available for the SPI and I2C bus only): Gets the last asynchronous read/write operation of return value.
- ADL_BUS_CMD_SET_ASYNC_MODE (Available for the SPI bus and I2C only): Configure the Synchronous/asynchronous mode settings.
- ADL_BUS_CMD_GET_ASYNC_MODE (Available for the SPI and I2C bus only): Gets the current value of the synchronous/asynchronous mode settings.
- ADL_BUS_CMD_SET_SPI_MASK_AND_SHIFT (Available for the SPI bus only): Enables/disables and set the parameters for the mask and shift modes.
- ADL_BUS_CMD_GET_SPI_MASK_AND_SHIFT (Available for the SPI bus only): Gets the status and the parameters for the mask and shift modes.
- ADL_BUS_CMD_SET_PARALLEL_CFG (Available for the Parallel bus only): Sets the Parallel configuration for one subscribed bus.
- ADL_BUS_CMD_GET_PARALLEL_CFG (Available for the Parallel bus only): Gets the Parallel configuration for one subscribed bus.
- ADL_BUS_CMD_PARA_GET_ADDRESS (Available for the Parallel bus only): Gets Parallel bus base where the chip select can be addressed for one subscribed bus.

- ADL_BUS_CMD_PARA_GET_MAX_SETTINGS (Available for the Parallel bus only): Gets Parallel bus maximum values.
- ADL_BUS_CMD_PARA_GET_MIN_SETTINGS (Available for the Parallel bus only): Gets Parallel bus minimum values.

Param: Parameter associated to the”Cmd” parameter.

Return Values

OK is returned on successful execution of the API.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_PARAM is returned if a parameter has an incorrect value.
ADL_RET_ERR_DONE is returned if an error occurs during the operation.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level Interrupt handler.

3.4. Reading Data from the Subscribed Bus (SPI and I2C)

The following API should be used to read data from a subscribed bus.

Prototype:

```
s32 adl_busRead( s32 Handle, adl_busAccess_t * pAccessMode, u32 Length, void * pDataToRead )
or
s32 adl_busReadExt( s32 Handle, adl_busAccess_t * pAccessMode, u32 Length, void * pDataToRead, void * Context )
```

Parameters:

Handle: Handle returned by the adl_busSubscribe function.
pAccessMode: Bus access mode, defined according to the following type:

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

This parameter is processed differently according to the bus types:

Address: This is usable for both SPI and I2C buses. The address parameter allows 32 bits to be sent on the bus, before the read and the write process is started. 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: This parameter can be used only over the SPI bus. The Opcode parameter allows sending 32 bits over the bus before the read or the write process. The number of bits to send is set by the

ADL_BUS_CMD_SET_OP_SIZE IOCtl 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).

For example: If "BBB" has to be sent over the bus the Opcode parameter has to be set to 0xBBB00000 value, and the OpcodeLength parameter has to be set to 12.

Length: Number of bytes to be read from the bus.

pDataToRead: Buffer to which the read bytes must be copied.

Context: Pointer to an application context, which will be provided back to the application when the asynchronous read operation end event will occur.

Return Values

OK is returned on successful execution of the API.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_PARAM is returned if a parameter has an incorrect value.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler in synchronous mode.

3.5. Reading Data from the Subscribed Parallel Bus

The following API is used to read from previously subscribed parallel bus.

Prototype:

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

Parameter:

Handle: Handle returned during the bus subscription.

ChipAddress: Chip address configuration.

DataLen: Number of items (16 bits or 8 bits set as bus width) to be read.

Data: Buffer where data will be copied.

Returned Value:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the handle is invalid.
ADL_RET_ERR_PARAM is returned if parameters are incorrect.

3.6. Writing Data to the Subscribed Bus (SPI and I2C)

The following API should be used to write data on a subscribed bus:

Prototype:

```
s8 adl_busWrite( u8 Handle, adl_busAccess_t * pAccessMode, u32 Length, void * pDataToWrite );
or
s8 adl_busWrite( u8 Handle, adl_busAccess_t * pAccessMode, u32 Length, void * pDataToWrite,
void * Context);
```

Parameters:

Handle: Handle returned by the adl_busSubscribe function.

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

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

This parameter is processed differently according to the bus types:

Address: This is usable for both SPI and I2C buses. The address parameter allows 32 bits to be sent on the bus, before the read and the write process is started. 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: This parameter can be used only over the SPI bus. The Opcode parameter allows sending 32 bits over the bus before the read or the write process. The number of bits to send is set by the ADL_BUS_CMD_SET_OP_SIZE IOCtl 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).
For example: If "BBB" has to be sent over the bus the Opcode parameter has to be set to 0xBBB00000 value, and the OpcodeLength parameter has to be set to 12.

Length: Number of bytes to be written on the bus.

pDataToRead: Data buffer to be written on the bus.

Context: Pointer to an application context, which will be provided back to the application when the asynchronous read operation end event will occur.

Returned Values:

OK is returned on successful execution of the API.

ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_PARAM is returned if a parameter has an incorrect value.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler in synchronous mode.

3.7. Writing Data to the Subscribed PARALLEL Bus

The following API is used to write to previously subscribed parallel bus.

Prototype:

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

Parameter:

Handle: Handle returned during the bus subscription.

ChipAddress: Chip address configuration.

DataLen: Number of item (16 bits or 8 bits set as bus width) to be written.

Data: Buffer from which data is copied.

Returned Value:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is incorrect.
ADL_RET_ERR_PARAM is returned incorrect parameter.

4. Sample Code for writing to SPI bus.

```
#include "adl_global.h"
#include "adl_bus.h"

// 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 ];

void MyFunction ( void )
{
  // Local variables
  s32 ReadValue;
  // 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, 0 );
  adl_busIOCtl ( MyI2CHandle, ADL_BUS_CMD_SET_ADD_SIZE, 0 );
  // 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 );
}

void adl_main ( adl_InitType_e InitType )
{
  MyFunction();
}
```

Summary

The following points have been covered in this chapter

- **Bus refers to a common data path or channel between multiple devices acting as peripherals.**
- **Embedded module support three buses – SPI bus, I2C bus and parallel bus.**
- **The APIs provided for bus management are defined in the adl_bus.h header file.**
- **SPI and I2C buses are serial buses.**
- **PARALLEL bus support is provided from Open AT OS v4.1x and AirPrime Q26XX embedded module.**
- **To subscribe to the bus service, use the adl_busSubscribe API.**
- **To unsubscribe from the bus service, use the adl_busUnsubscribe API.**
- **To read data from a subscribed SPI/I2C bus, use the adl_busRead API. To read data from a parallel bus, use adl_busDirectRead API.**
- **To write data to a subscribed SPI/I2C bus, use the adl_busWrite API. To write data to a parallel bus, use adl_busDirectWrite API.**

CHAPTER 20

Watchdog

1. Objective

This chapter describes the watchdog service and its usage in the embedded module. Open AT OS provides a watchdog service to access to the embedded module hardware watchdog timer.

This chapter addresses the following questions:

- What is Watchdog timer?
- Different types of Watchdog timer?
- Different Watchdog service provided by ADL?

APIs related to the watchdog service are declared in the adl_wd.h header file.

2. Watchdog Timer

A watchdog timer is a hardware timer that triggers a system reset in the main program, due to some fault conditions, such as a hang. The reset is done in order to bring the system back to normal operation state. The most common use of watchdog timers is in embedded systems, where this specialized timer is often a built-in unit of a microcontroller.

Watchdog timers may also trigger control systems to move into a safety state, such as turning off motors, high-voltage electrical outputs, and other potentially dangerous subsystems until the fault is cleared.
ADL provides APIs to disable/enable hardware watchdog and also to start/stop software watchdog.

Open AT OS provides two types of Watchdog timers:

- Hardware Watchdog Timer
- Software Watchdog Timer

2.1. Hardware Watchdog Timer

Open AT OS maintains a hardware timer for stability of all running tasks in the system. These tasks include Firmware tasks as well as Open AT Application tasks.

The lowest priority task re-starts the Watchdog timer when it is being scheduled. If the lowest priority task is not scheduled in 5 seconds, the timer expires and causes a system reset. This reset is done in order to avoid any tasks hogging the embedded module for long time. For e.g. a task in infinite loop.

Enable/Disable of Hardware Watchdog

Open AT OS provides APIs which can be used for disabling the hardware watchdog for a specified duration. This avoids the embedded module restart when Open AT Application takes more than the hardware watchdog duration. This feature can be used in the Open AT Application which has operations which takes more than hardware watchdog time. Ex: SSL Applications.

Following is the list of APIs that provide this functionality:

- adl_wdPut2Sleep
- adl_wdAwake

These APIs are described in more detail in section 5.

2.2. Software Watchdog Timer

Open AT OS maintains a software watchdog timer for Open AT Application tasks. Open AT Application task can start this timer and reset it before the expiry. If Open AT Application task is unable to reset the timer in specified time, timer will expire and it will cause a reset. This mechanism will ensure that all Open AT Application tasks are being scheduled and the system is stable. In case of instability, reset will ensure the stability.

Start/Stop of Application Watchdog

ADL provide APIs which can be used to start and stop the software watchdog from an Open AT Application. Once the software watchdog is started it should be restarted again to avoid Watchdog reset.
Following list of APIs provide this functionality:

- adl_wdActiveAppWd
- adl_wdRearmAppWd
- adl_wdDeActiveAppWd

These APIs are described in more detail in section 5.

3. Advantages of Watchdog Timer

Following are the advantages of Watchdog Timer:

- Prevents the embedded module from entering-the blocking state.
- Prevents the Open AT Application from entering a ‘hang’ state.
- Allows user to overcome system instability.

4. Restrictions & Limitations of Watchdog Timer

Following are the limitations of Watchdog Timer:

- When watchdog is declared in one of the Open AT Application tasks and if another task declares it again, the last Watchdog timer value is overwritten.
- When delaying the Hardware watchdog, the GSM stack will still be pre-empting the Open AT Application if needed. It is strongly advised to calibrate the delay using the worst case scenario (GPRS or CSD transfers ongoing).
- When delaying the HW watchdog the application watchdog shall still be re-armed regularly. Either increase temporarily the Application watchdog for the time of the delay, or do the “heavy” treatment in a less priority task than the one re-arming the Application watchdog.

5. Watchdog Timer APIs

5.1. Deactivation of Hardware Watchdog

This API is used to deactivate the hardware watchdog for a specified duration. During the watchdog sleep duration, Open AT Application treatments can take more time than the hardware watchdog duration. Once the sleep duration is expired, the hardware watchdog will be activated again.

Prototype:

s8 adl_wdPut2Sleep (u32 i_u32_SleepDuration)

Parameters:

i_u32_SleepDuration: This parameter indicates the watchdog sleep duration in number of ticks.
Returned Values:
This function returns OK on success or ADL_RET_ERR_PARAM error code if wrong argument is used.

5.2. Reactivation of Hardware Watchdog

This API is used to reactivate the hardware watchdog. If this API is not called, the default behavior of the hardware watchdog will be restored automatically at the expiration of watchdog sleep duration.

Prototype:

s8 adl_wdAwake (void)

Parameters:

None.

Returned Values:

This function returns remaining time before automatic watchdog reactivation in number of ticks.

5.3. Activation of Application Watchdog

This API is used to activate the Open AT Application watchdog. Open AT Application watchdog must be rearmed regularly to indicate that it is still alive. embedded module resets if watchdog timer expires.

Prototype:

s32 adl_wdActivateAppWd (u32 i_u32_Duration)

Parameters:

i_u32_Duration: This parameter indicates the watchdog sleep duration in number of ticks.

Returned Values:

This function returns OK on success or ADL_RET_ERR_NOT_SUPPORTED if watchdog service is not supported.

5.4. Rearm of Application Watchdog

This API is used to rearm the application watchdog. This API returns OK as response even if application watchdog is not started before. The cyclic timer can be used to rearm the application watchdog.

Prototype:

s32 adl_wdRearmAppWd (void)

Parameters:

None.

Returned Values:

This function returns OK on success or ADL_RET_ERR_NOT_SUPPORTED if watchdog service is not supported.

5.5. Deactivation of Application Watchdog

This API is used to deactivate the application watchdog

Prototype:

s32 adl_wdDeActiveAppWd (void)

Parameters:

None.

Returned Values:

This function returns OK on success or ADL_RET_ERR_NOT_SUPPORTED if watchdog service is not supported.

6. Sample Code

```
#include "adl_global.h"
#include "adl_wd.h"
const u16 wm_apmCustomStackSize = 4096;

u8 wd_command = 0;
bool heavy_treatment_done = FALSE;
u32 counter1;

void TimerHandler( u8 ID, void* context )
{
 TRACE((1,"Timer handler is called"));
 switch(wd_command)
 {
  case 1:
   TRACE((1,"Cmd: 1....No reset due to HW WD"));
   break;
  case 2:
   TRACE((1,"Cmd: 2....No reset due to HW WD"));
   break;
  case 3:
   TRACE((1,"Cmd: 3....No reset due to HW WD"));
   break;
  case 4:
   {
    s32 ret;
    counter1--;
    if (counter1==0)
    {
     TRACE((1,"Cmd: 6....No reset due to HW WD"));
    }
    else if (counter1==1)
    {
     TRACE((1,"Cmd: 6....Deactive appli WD, verify NO reset"));
     adl_wdDeActiveAppWd();

     // launch timer to test result
     adl_tmrSubscribe(FALSE,200,ADL_TMR_TYPE_100MS,TimerHandler);
    }
    else
    {
     TRACE((1,"Cmd: 6....rearm appli WD"));
     adl_wdRearmAppWd();

     // launch timer to test result
```

```
        adl_tmrSubscribe(FALSE,90,ADL_TMR_TYPE_100MS,TimerHandler);
      }
    }
    break;
  }
}


//void HeavyTreatment( u8 ID, void *context )
void HeavyTreatment( void )
{
  u32 counter;

  TRACE((1,"Heavy Treatment started"));
  counter=0x05FFFFFF; //(env 20s)
  while(counter--);
  heavy_treatment_done = TRUE;
}

static void WD_Handler( adl_atCmdPreParser_t * paras )
{

  wd_command = wm_atoi( ADL_GET_PARAM( paras, 0 ) );
  switch( wd_command )
  {
    case 1:
      {
        u32 ret;
        /* Use the function adl_wdPut2Sleep - No reset */
        /* WD sleep time and heavy treatment execution time are same */
        TRACE((1,"sleep HW WD 20s "));
        ret = adl_wdPut2Sleep(ADL_TMR_S_TO_TICK(20));

        TRACE((1,"Return value of adl_wdPut2Sleep = %d", ret));
        // launch timer to test result
        TRACE((1,"verify reset in ~35s "));
        adl_tmrSubscribe(FALSE,350,ADL_TMR_TYPE_100MS,TimerHandler);

        TRACE((1,"launch heavy treatment ~ 20s"));
        // launch heavy treatment
        //adl_tmrSubscribe(FALSE,10,ADL_TMR_TYPE_100MS,HeavyTreatment);
        HeavyTreatment();
      }
      break;

    case 2:
      {
        u32 ret;
        /* Use the function adl_wdPut2Sleep - Reset due to WD*/
        /* WD sleep time is less than the  heavy treatment execution time */
        TRACE((1,"sleep HW WD 10s "));
        ret = adl_wdPut2Sleep(ADL_TMR_S_TO_TICK(10));
        TRACE((1,"Return value of adl_wdPut2Sleep = %d", ret));

        // launch timer to test result
        TRACE((1,"verify reset in ~35s "));
        adl_tmrSubscribe(FALSE,350,ADL_TMR_TYPE_100MS,TimerHandler);

        TRACE((1,"launch heavy treatment ~ 20s"));
        // launch heavy treatment
        //adl_tmrSubscribe(FALSE,10,ADL_TMR_TYPE_100MS,HeavyTreatment);
        HeavyTreatment();
      }
      break;
    case 3:
      {
        u32 ret;
        /* Use the function adl_wdPut2Sleep - No Reset due to WD*/
        /* Heavy treatment execution time is lesser than WD sleep time */
        TRACE((1,"sleep HW WD 30s "));
        ret = adl_wdPut2Sleep(ADL_TMR_S_TO_TICK(30));

        TRACE((1,"Return value of adl_wdPut2Sleep = %d", ret));
        // launch timer to test result
        TRACE((1,"verify reset in ~35s "));
        adl_tmrSubscribe(FALSE,350,ADL_TMR_TYPE_100MS,TimerHandler);

        TRACE((1,"launch heavy treatment ~ 20s"));
        // launch heavy treatment
        //adl_tmrSubscribe(FALSE,10,ADL_TMR_TYPE_100MS,HeavyTreatment);
        HeavyTreatment();
      }
      break;
    case 4:
      {
        //Activate a watchdog with adl_wdActiveAppWd () with a duration.
        //Rearm it with adl_wdRearmAppWd() before duration expires
        //Deactivate it with adl_wdDeActiveAppWd() before duration expires
        //Verify you have NO reset.
        s32 ret;

        counter1 = 5;
        // Lets activate the application watchdog for 30 seconds
        ret = adl_wdActiveAppWd(ADL_TMR_S_TO_TICK(10));
```

```
            // Lets suscribe to a 25 sec timer
            TRACE((1,"verify reset in ~25sec "));
            adl_tmrSubscribe(FALSE,90,ADL_TMR_TYPE_100MS,TimerHandler);

           }
           break;
         default:
           break;
       }
     }

     void adl_main (adl_InitType_e adlInitType)
     {
       TRACE((1,"Main function"));
        adl_atCmdSubscribe ( "AT+WD",WD_Handler, ADL_CMD_TYPE_PARA | 0x11 );
     }
```

Summary

The following points have been covered in this chapter

- **ADL provides Watchdog APIs to stop/start the hardware watchdog and also the application watchdog.**
- **The adl_wdPut2Sleep API should be used to deactivate the hardware watchdog.**
- **The adl_wdAwake API should be used to resume the hardware watchdog.**
- **The adl_wdActivateAppWd API should be used to activate the application watchdog.**
- **The adl_wdRearmAppWd API should be used to rearm the application watchdog.**
- **The adl_wdDeActiveAppWd API should be used to deactivate the application watchdog.**

CHAPTER 21

SIM Service

1. Objective

This chapter introduces the SIM service and its related APIs. The various SIM- and PIN-related events are explained here.

2. Overview of the SIM Service

Open AT OS provides the SIM service to handle all the SIM and PIN related functionality. SIM service can be used in the following situations:

- When the SIM is inserted and removed
- When the SIM PIN is required to be verified to start any SIM-related events

3. Events Related to the SIM and PIN

SIM- and PIN-related events like SIM PIN required, SIM inserted or SIM removed are indications sent by the SIM to the embedded module.

Following is list of SIM- and PIN-related events and their corresponding numeric values:

SIM Events	Numeric Value	Description
ADL_SIM_EVENT_REMOVED	1	This event is received when the SIM card is removed from embedded module.
ADL_SIM_EVENT_INSERTED	2	This event is received when the SIM card is inserted into the Embedded module.
ADL_SIM_EVENT_FULL_INIT	3	This event is received when the SIM card is initialized completely or WIND: 4 has been received by Open AT Application. After receiving this event any SIM-based AT commands such as phonebook command or SMS command can be executed.

SIM Events	Numeric Value	Description
ADL_SIM_EVENT_PIN_ERROR	4	This event is received when the incorrect PIN code is entered.
ADL_SIM_EVENT_PIN_OK	5	This event is received when the correct PIN code is entered.
ADL_SIM_EVENT_PIN_WAIT	6	This event is received when the PIN is required to be sent to the SIM or if the PIN field in the adl_simSubscribe API is set to NULL (refer to the section on Subscription to SIM service).
ADL_SIM_EVENT_PIN_NO_ATTEMPT	7	This event is received when only one attempt is left to enter the right PIN code.
ADL_SIM_EVENT_LAST	8	This is last SIM state. It is never set. It has been defined for the programming purpose.

4. SIM States

SIM State describes the current state of the SIM – whether it is inserted or removed and so on. A SIM can be in only one state at any given instant. Based on the SIM state, the embedded module can decide which functions can be used at that time. For example if SIM state is set as ADL_SIM_STATE_FULL_INIT (defined below) then all the SIM related commands could be processed.

SIM state is set automatically by the embedded module whenever a SIM-related event occurs. SIM states have been defined as enumerated data type of type "adl_simState_e" as shown below.

```
typedef enum
{
   ADL_SIM_STATE_INIT,
   ADL_SIM_STATE_REMOVED,
   ADL_SIM_STATE_INSERTED,
   ADL_SIM_STATE_FULL_INIT,
   ADL_SIM_STATE_PIN_ERROR,
   ADL_SIM_STATE_PIN_OK,
   ADL_SIM_STATE_PIN_WAIT,
   ADL_SIM_STATE_LAST
} adl_simState_e;
```

Following is list of SIM states, their corresponding numeric value and name in Open AT OS.

SIM Events	Numeric Value	Description
ADL_SIM_STATE_INIT	0	This SIM state is set when the SIM is initializing. At this time SIM PIN state is not known to the Open AT OS
ADL_SIM_STATE_REMOVED	1	This state is set when the SIM is removed from the embedded module.
ADL_SIM_STATE_INSERTED	2	This state is set when the SIM is inserted into the embedded module.
ADL_SIM_STATE_FULL_INIT	3	This state is set when the SIM is completely initialized or WIND: 4 is received by Open AT OS.
ADL_SIM_STATE_PIN_ERROR	4	This state is set when the incorrect PIN code is issued.
ADL_SIM_STATE_PIN_OK	5	This state is set when the correct PIN code is issued. After issuing the correct PIN code, the Open AT OS waits for the complete initialization.
ADL_SIM_STATE_PIN_WAIT	6	This state is set when the SIM is inserted into the embedded module but the PIN code is not issued yet.
ADL_SIM_STATE_LAST	7	This is last SIM state. It is never set. It has been defined for the programming purpose.

5. API Related to the SIM Service

Following is the list of APIs which are used to handle SIM-related events.

- adl_simSubscribe
- adl_simUnsubscribe
- adl_simGetState

Include the adl_sim.h header file to use the SIM service.

5.1. Subscription to SIM service

The following API should be used to subscribe to SIM service:

Prototype:

```
s32 adl_simSubscribe (adl_simHdlr_f Handler, ascii * PinCode)
```

This API subscribes to SIM service and specifies a callback function which is called whenever SIM- or PIN-related events are received. It also allows you to enter PIN code of the SIM.

Parameters:

Handler: Handler is the callback function which is called whenever a SIM or PIN event is received by the Open AT Application. has the following declaration:

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

The input parameter to this callback function is a SIM event. Refer to the section on Events Related to the SIM and PIN.

PinCode: PinCode is defined as ASCII * and is the string containing the PIN code of the SIM. If it is set to NULL or if the provided code is incorrect, then the PIN code should be entered by the external application. This argument is used only the first time the SIM service is subscribed.

Returned Value:

OK is returned on success.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level Interrupt handler.
ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the service was already subscribed with the same handler.
ADL_RET_ERR_PARAM is returned if the function was called with a null handler.

5.2. Unsubscribe from the SIM Service

The following API should be used to unsubscribe from the SIM service:

Prototype:

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

You can only unsubscribe from a subscribed SIM service.

Parameters:

Handler: This parameter is the callback function earlier passed to the adl_simSubscribe API.

Returned Value:

OK is returned on success.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level Interrupt handler.

5.3. Getting the SIM State

Use the following API to get the current state of the SIM:

Prototype:

```
adl_simState_e adl_simGetState ( void )
```

This state is set by the Open AT OS whenever a SIM-related event occurs.

Parameters:

None.

Returned Values:

This API returns the current SIM states (refer to the section on SIM States).

NOTE :
The SIM service can be subscribed several times in Open AT Application.
If SIM service is subscribed multiple times, and any event occurs, then all the handlers will be called by the Open AT OS.
You can only unsubscribe a subscribed handler of the SIM service, or an error will be returned.
The SIM be in only one state at any point of time.

5.4. Entering a new PIN code

This API can be used to enter a PIN code of the inserted SIM.

Prototype:

```
s32 adl_simEnterPIN ( ascii * PinCode )
```

This state is set by the Open AT OS whenever a SIM-related event occurs.

Parameters:

PinCode: This parameter holds the PIN code.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if the Pin Code is not informed.
ADL_RET_ERR_BAD_STATE is returned if the SIM is not waiting for any Pin Code to be entered.

5.5. Entering PUK code

This API can be used to enter PUK code and a new pin code.

Prototype:

```
s32 adl_simEnterPUK ( ascii * PukCode, ascii* NewPinCode )
```

Parameters:

PukCode: This parameter holds the PUK code.

NewPinCode: This parameter holds the new PIN code.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned 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.

5.6. Get the remaining attempts on PIN and PUK codes

This API can be used to get the remaining number of 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;
```

6. Sample Code

```
/* Mandatory header file to be included to use SIM service */
#include adl_sim.h
/* Global variable is declared to capture the SIM state */
adl_simState_e simState;
/* This function is called whenever any SIM event is received */
void simHandler (u8 Event)
{
                /* Here received SIM event has been handled */
switch (Event)
{
case ADL_SIM_EVENT_INSERTED:
adl_atSendResponse (ADL_AT_UNS,"\r\SIM is inserted\r\n");
break;
case ADL_SIM_EVENT_REMOVED:
adl_atSendResponse (ADL_AT_UNS,"\r\SIM is removed\r\n");
break;
}
}
/* Open AT entry point function */
void adl_main (adl_InitType_e adlInitType)
{
/* subscribe for SIM service, whose handler is simHandler and PIN */ /* is given as 0000*/
adl_simSubscribe (simHandler, "0000")
}
```

Summary

The following points have been covered in this chapter

- **Open AT OS provides the SIM service to handle SIM-related events.**
- **SIM events are sent by the SIM to the embedded module.**
- **To subscribe a SIM service, use the adl_simSubscribe function.**
- **To unsubscribe a SIM service, use the adl_simUnSubscribe function.**
- **To get the current SIM state, use the adl_simGetState function**

CHAPTER 22

Short Message Service

1. Objective

This chapter describes the SMS (Short Message Service) Service used to implement the exchange of short messages between subscribers in a GSM network. The SMS service provides a means of sending messages of limited size to and from GSM/UMTS mobiles. SMS can be classified on basis of:

Origin:

- Mobile-originated messages are transported from an MS (Mobile Station) to a service centre
- Mobile-terminated messages are transported from a service centre to an MS.

Data Mode:

- Text Mode: All commands and responses are in ASCII format.
- PDU mode: PDU stands for Protocol Data Unit. In this mode a complete SMS message including all header information is given as a binary string (in hexadecimal format).

2. What is SMS?

The SMS provides a means to transfer short messages between a GSM/UMTS MS and an SME (Short Message Entity) via a SC (Service Centre). The SC serves as a relaying station of the message transfer between the MS and the SME.

3. Basic Services

The Short Message Service comprises two basic services:

- SMMT (Short Message Mobile Terminated)
- SMMO (Short Message Mobile Originated)
- SMMT denotes the capability of the GSM/UMTS system to transfer an SMS submitted from the SC to one MS, and to provide information about the delivery of the SMS either by a delivery report or a failure report, with a specific mechanism for later delivery.

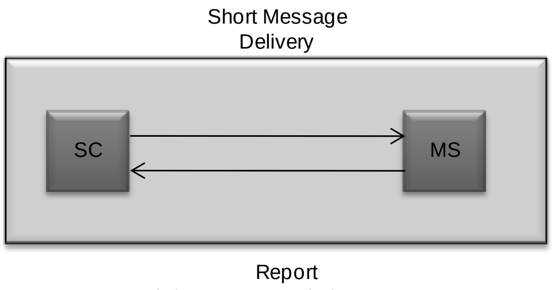


Figure 70 - Mobile Terminated Short Message Service

SMMO denotes the capability of the GSM/UMTS system to transfer an SMS submitted by the MS to one SME via an SC, and to provide information about the delivery of the SMS either by a delivery report or a failure report. The message must include the address of that SME to which the SC shall eventually attempt to relay the short message.

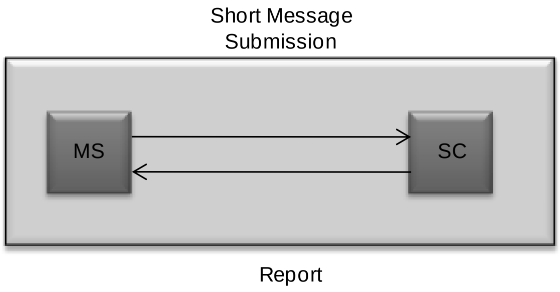


Figure 71 - Mobile Originated Short Message Service

The text message to be transferred by means of SMMT or SMMO can contain up to 140 bytes.

4. SMS Service Implementation

Open AT supports standard APIs which allow the SMS functionality to be implemented conveniently. These functions have been declared in the adl_sms.h header file.

The use of any service requires three basic steps:

- Subscription to the service
- Usage of service functions to implement functionality
- Un-subscribe from the service

4.1. Subscribing to the SMS Service

The following API should be used to subscribe and use the SMS service:

Prototype:

s8 adl_smsSubscribe (adl_smsHdlr_f smsHandler, adl_smsCtrlHdlr_f smsCtrlHandler, u8 Mode)

This API subscribes to the SMS service providing two associated callback functions:

Parameters:

smsHandler: smsHandler is called each time an SMS is received from the network. The handler is declared:

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

Parameters:

- SmsTel: This parameter contains the telephone number of the originating MS in text mode and NULL in PDU mode.
- SmsTimeLength: This parameter contains the SMS time stamp in text mode and SMS length in PDU mode.
- SmsText: This parameter contains the actual SMS text (in text mode), or the SMS PDU (in PDU mode). PDU stands for Protocol Data Unit and contains the SMS content along with required header information as a part of it.

Returned Values:

- TRUE: if the SMS must be forwarded to the external application. 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. This is called the SMS Read operation and the SMS is then stored in the SIM memory. The external application is notified by sending an unsolicited code: +CMTI:"SM",<>.
- FALSE: in case the SMS should not be forwarded to the network, this function sends False as a return value.

smsCtrlHandler: SMS control handler captures the events received during SMS sending process. The function type is defined below:

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

Parameters:

- Nb: This parameter takes different values depending upon the event received in SMS control handler.
- Event: Event received while sending SMS include:

Event	Description	Nb Parameter
ADL_SMS_EVENT_SENDING_OK	This event notifies that the SMS is sent successfully	Nb parameter value not relevant
ADL_SMS_EVENT_SENDING_ERROR	Error occurred when sending the SMS	Nb parameter takes the error number of the CMS error(+CMS ERROR")
ADL_SMS_EVENT_SENDING_MR	SMS sent successfully. This event is received along with ADL_SMS_EVENT_SENDING_OK event.	Nb parameter contains the sent Message Reference value.

- Mode: This parameter is used to specify which mode was used to send the SMS. As described above there are two modes:
- ADL_SMS_MODE_PDU: This parameter indicates that the smsHandler should be called in PDU mode on the receipt of each SMS.
- ADL_SMS_MODE_TEXT: This parameter indicates that the smsHandler should be called in Text mode on the receipt of each SMS.

Returned Values:

Positive or null handle is returned in case of successful subscription on SMS service. This handle is required for further SMS operations.
ADL_RET_ERR_PARAM is returned in case of wrong parameter is specified.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

4.2. Sending a SMS

The SMS service provides the adl_smsSend () function to send an SMS. This function must be invoked each time an SMS needs to be sent:

Prototype:

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

Parameters:

Handle: This is the handle returned by the adl_smsSubscribe function
SmsTel: This parameter holds the telephone number of the recipient MS (in Text mode) OR Null (in PDU mode).
SmsText: This parameter contains the SMS text in text mode or the SMS PDU in PDU mode.
Mode: This parameter is used to select between the Text or PDU SMS modes. The two possible values are:

ADL_SMS_MODE_PDU: to send SMS in PDU mode.

ADL_SMS_MODE_TEXT: to send SMS in Text mode.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned in case of incorrect parameter value is specified.
ADL_RET_ERR_UNKNOWN_HDL is returned if the handle provided to the adl_Send () function is different from the actual handle returned by the adl_sms_Subscribe () function.
ADL_RET_ERR_BAD_STATE is returned if due to some reason the embedded module is not ready to send the SMS. For instance, if the initialization is not done yet, or if the process of sending the SMS is already in progress.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interruption handler.

4.3. Unsubscribing from SMS service

The SMS service must be stopped by unsubscribing from the service:

Prototype:

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

Parameters:

Handle: This parameter is the handle returned by the earlier call to the adl_smsSubscribe API.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is not correct. The handle provided to adl_smsUnsubscribe () function should be same as that returned by the adl_smsSubscribe () function.
ADL_RET_ERR_NOT_SUBSCRIBED is returned if the subscription fails and the user tries to unsubscribe from the service.
ADL_RET_ERR_BAD_STATE is returned in case the SMS send process is in progress and the Open AT Application executes the unsubscribe function.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

5. Sample Code

```
/* sample code implementing the SMS send functionality */
#include "adl_global.h"
#include "adl_sms.h"
const u16 wm_apmCustomStackSize = 4096;

// Local variables
u8 Event;
u16 Nb;
s8 smshandle;
//ascii telno[]="<Telephone No.>";
ascii telno[]="9844472947";
ascii *smstimelnth;
ascii *smstext="The text for sms goes here";
s8 sendhandle;
s8 unshandle;
//Local Functions
bool SmsHandler(ascii * telno, ascii *smstimelnth,ascii *smstext)
{
  adl_atSendResponse(ADL_AT_UNS,"inside sms hsndler");
  return (0);
}

void Timerhdl(u8 id)
{
  TRACE (( 1, "Unsubscribing the SMS" ));
  adl_smsUnsubscribe(smshandle);
}
/* sms control handler captures the events received on SMS sending*/
void SmsCtrlHandler(u8 Event,u16 Nb)
{
  s8 sRet;
  switch(Event)
  {
    case ADL_SMS_EVENT_SENDING_OK:
      adl_atSendResponse(ADL_AT_RSP,"SMS Sent Successfully");
      TRACE((1,"Inside ADL_SMS_EVENT_SENDING_OK EVENT"));
      /* if the SMS send is successful, subscribe to timer of short duration and in
        the timer handler unsubscribe from the SMS service.*/
      adl_tmrSubscribe(FALSE, 10, ADL_TMR_TYPE_100MS, (adl_tmrHandler_t) Timerhdl);
      TRACE (( 1, "SMS sent successfully, now unsubscribe" ));
      break;
    case ADL_SMS_EVENT_SENDING_ERROR:
      adl_atSendResponse(ADL_AT_RSP,"error sending sms");
      sRet = adl_smsSend(smshandle,telno,smstext,ADL_SMS_MODE_TEXT);
      TRACE (( 1, "error sending sms, try again" ));
      break;
    case ADL_SMS_EVENT_SENDING_MR:
      TRACE (( 1, "sms send successful" ));
      TRACE((1,"Inside ADL_SMS_EVENT_SENDING_MR EVENT"));
      break;
    default:
      adl_atSendResponse(ADL_AT_UNS,"Inside default");
      break;
  }
}
bool wind_4_handler(adl_atUnsolicited_t * paras)
{
  s8 sRet=0;
  TRACE((1,"Inside wind 4 handler"));
  /* Use adl_smsSend () function  to send SMS*/
  /* telephone number of the recipient and SMS text are defined in the   variables above*/
  sRet = adl_smsSend(smshandle,telno,smstext,ADL_SMS_MODE_TEXT);
  return (0);
}
// Main function
void adl_main ( adl_InitType_e InitType )
{
  TRACE (( 1, "Embedded Application: Main" ));
  /* Subscribe to SMS service to implement SMS services*/
  smshandle=adl_smsSubscribe((adl_smsHdlr_f)SmsHandler,
                    (adl_smsCtrlHdlr_f)SmsCtrlHandler,ADL_SMS_MODE_TEXT);
  /* subscribe to wind4 unsolicited response in order to wait for wind4
     response to occur*/
  adl_atUnSoSubscribe("+WIND: 4",wind_4_handler);
}
```

NOTE :
The SMS service must be subscribed before using any of SMS service API.
The adl_smsSend () function must be called after WIND 4 indication. For this adl_atUnSoSubscribe () function can be used to subscribe to WIND 4 unsolicited response and further functionality can be implemented in the Handler of this function. This is because the embedded module is able to accept all commands after WIND 4 indication.
At the end of Open AT Application, the SMS service must be unsubscribed.
In case of dynamic memory allocation, ensure that the allocated memory is released at the end of program.

Summary

The following points have been covered in this chapter

- **SMS is a means of sending messages of limited size to and from GSM/UMTS mobiles. While writing an Open AT Application for the SMS service, ensure that the format supported for different languages is determined. The two SMS format modes available are Text and PDU modes.**
- **The SMS service is started by subscribing to adl_smsSubscribe () function. The adl_smsSubscribe () provides two handlers:**
- **DataHandler: to capture data elements received when a SMS is received from the network.**
- **ControlHandler: to capture the events received when an SMS is sent. These events tell the status of SMS send/receive function. The ADL_SMS_EVENT_SENDING_ERROR event gives the error code returned.**
- **The adl_smsSend () function is used to send an SMS. It takes the necessary parameters (telephone number, SMS text and SMS mode (Text or PDU)) required to send an SMS.**
- **In the end adl_smsUnsubscribe () function must be used to unsubscribe from the service. Unsubscribing from a service means no more events would be received by the Open AT Application.**

CHAPTER 23

Call Service

1. Objective

This chapter introduces you to the call service and its procedures for:

- Incoming and outgoing calls
- Answering calls
- Hanging up (terminating) calls

Various call-related events and APIs are also explained in this chapter.

2. Call Service Events

Call service events are received by the Open AT Application when any incoming or outgoing call is detected by the embedded module. These events are issued by the embedded module to the Open AT Application.

Following is a list of events related to the call service and their corresponding numeric values and descriptions:

Call Events	Description
ADL_CALL_EVENT_RING_VOICE	The embedded module has received the voice call.
ADL_CALL_EVENT_RING_DATA	The embedded module has received the data call.
ADL_CALL_EVENT_NEW_ID	The embedded module has received the WIND: 5,x indication. x can have any value.
ADL_CALL_EVENT_RELEASE_ID	The embedded module has received the WIND: 6,x indication. x can have any value.
ADL_CALL_EVENT_ALERTING	The embedded module has received the WIND: 2 indication.
ADL_CALL_EVENT_NO_CARRIER	The Embedded module has received the NO CARRIER indication.
ADL_CALL_EVENT_NO_ANSWER	The embedded module has received the NO ANSWER indication.

Call Events	Description
ADL_CALL_EVENT_BUSY	The embedded module has received the BUSY indication.
ADL_CALL_EVENT_SETUP_OK	The embedded module has received the OK indication when the call was set up from the Open AT Application.
ADL_CALL_EVENT_ANSWER_OK	The incoming call is accepted by the Open AT Application and incoming call indication is not sent to external application.
ADL_CALL_EVENT_CIEV	The embedded module has received the OK indication when the call was set up from the Open AT Application.
ADL_CALL_EVENT_HANGUP_OK	The incoming call is terminated by the Open AT Application and the incoming call indication is not sent to external application.
ADL_CALL_EVENT_SETUP_OK_FROM_EXT	The ATD command was issued by external application and the Open AT Application has received the OK indication in response to this command.
ADL_CALL_EVENT_ANSWER_OK_FROM_EXT	The ATA command was issued by external application and the Open AT Application has received the OK indication in response to this command.
ADL_CALL_EVENT_HANGUP_OK_FROM_EXT	The ATH command was issued by external application and the Open AT Application has received the OK indication response to this command.
ADL_CALL_EVENT_AUDIO_OPENNED	The embedded module has received the +WIND: 9 indication.
ADL_CALL_EVENT_ANSWER_OK_AUTO	The embedded module has automatically accepted the incoming call (ATS0 is set to non- zero value).
ADL_CALL_EVENT_RING_GPRS	The embedded module has received the GPRS call.
ADL_CALL_EVENT_SETUP_FROM_EXT	The ATD command is issued by external application.
ADL_CALL_EVENT_SETUP_ERROR_NO_SIM	The Open AT Application or external application attempts to set up a call and fails due to the absence of the SIM card.
ADL_CALL_EVENT_SETUP_ERROR_PIN_NOT_READY	The Open AT Application or external application attempts to set up a call and fails because the PIN code of the SIM card was not entered.

Call Events	Description
ADL_CALL_EVENT_SETUP_ERROR	The Open AT Application or external application attempts to set up a call and fails for any other reason other than the above reasons.

If the call service is subscribed to multiple times using the function adl_callSubscribe, and the ADL_CALL_EVENT_SETUP_FROM_EXT event is received by the call handler function, then the following conditions can occur:
Case 1: If all the handlers return ADL_CALL_FORWARD then call set up is performed.
Case 2: If any of the handlers return ADL_CALL_NO_FORWARD or ADL_CALL_NO_FORWARD_ATH or ADL_CALL_NO_FORWARD_ATA constant, then CME ERROR: 600 is sent to the external application.

3. Call Service APIs

Following is the list of APIs which are used to handle the events related to the call service:

adl_callSubscribe
adl_callSetup
adl_callSetupExt
adl_callHangup
adl_callHangupExt
adl_callAnswer
adl_callAnswerExt
adl_callUnsubscribe

Call service APIs are declared in the adl_call.h header file.

3.1. Subscription to Call Service

Use the following API to subscribe to the call service:

Prototype:

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

This API subscribes to call service and specifies a callback function which is called whenever events related to the call service are received.

Parameters:

CallHandler: the callback function which is called whenever an event related to the call is received by the Open AT Application. The Handler is defined as the following function pointer:

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

Event: An event related to the call service. These events are explained in the section on Events Related to the Call Service.

Call_Id: The Identifier associated with the event. The following table lists the call events and their corresponding Call_IDs:

Call Events	Call Id
ADL_CALL_EVENT_RING_VOICE	0
ADL_CALL_EVENT_RING_DATA	0
ADL_CALL_EVENT_NEW_ID	The call identification number of the corresponding +WIND: 5 indication.
ADL_CALL_EVENT_RELEASE_ID	The call identification number of the corresponding +WIND: 6 indication.
ADL_CALL_EVENT_ALERTING	0
ADL_CALL_EVENT_NO_CARRIER	0
ADL_CALL_EVENT_NO_ANSWER	0
ADL_CALL_EVENT_BUSY	0
ADL_CALL_EVENT_SETUP_OK	For voice calls, the value of the call ID is 0. For data calls, the value of the call ID is the speed of the data call.
ADL_CALL_EVENT_ANSWER_OK	For voice calls, the value of the call ID is 0. For data calls, the value of the call ID is the speed of the data call.
ADL_CALL_EVENT_CIEV	For voice calls, the value of the call ID is 0. For data calls, the value of the call ID is the speed of the data call.
ADL_CALL_EVENT_HANGUP_OK	For voice calls, the value of the call ID is 0. For data calls, the value of the call ID is the ADL_CALL_DATA_FLAG constant.
ADL_CALL_EVENT_SETUP_OK_FROM_EXT	For voice calls, the value of the call ID is 0. For data calls, the value of the call ID is the speed of the data call.
ADL_CALL_EVENT_ANSWER_OK_FROM_EXT	For voice calls, the value of the call ID is 0. For data calls, the value of the call ID is the speed of the data call.
ADL_CALL_EVENT_HANGUP_OK_FROM_EXT	For voice calls, the value of the call ID is 0. For data calls, the value of the call ID is the ADL_CALL_DATA_FLAG constant.
ADL_CALL_EVENT_AUDIO_OPENNED	0

Call Events	Call Id
ADL_CALL_EVENT_ANSWER_OK_AUTO	For voice calls, the value of the call ID is 0. For data calls, the value of call ID is the speed of the data call.
ADL_CALL_EVENT_RING_GPRS	0
ADL_CALL_EVENT_SETUP_FROM_EXT	For voice calls, the value of the call ID is 0. For GSM data calls, the value of call ID is the ADL_CALL_DATA_FLAG constant. For GPRS calls, the value of the call ID is the binary OR of the ADL_CALL_GPRS_FLAG constant and the active call identifier.
ADL_CALL_EVENT_SETUP_ERROR_NO_SIM	0
ADL_CALL_EVENT_SETUP_ERROR_PIN_NOT_READY	0
ADL_CALL_EVENT_SETUP_ERROR	The value of the call ID is the CME ERROR value.

Following is the list of possible return values of the call handler and their corresponding descriptions:

Call handler return value	Value	Description
ADL_CALL_FORWARD	0	The event related to the call service must be forwarded to the external application.
ADL_CALL_NO_FORWARD	1	The event related to the call service is not forwarded to the external application.
ADL_CALL_NO_FORWARD_ATH	2	The event related to the call service is not forwarded to the external application. The incoming call is terminated by the Open AT Application by sending an ATH command.
ADL_CALL_NO_FORWARD_ATA	3	The event related to the call service is not forwarded to the external application. The incoming call is accepted by the Open AT Application by sending an ATA command.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned when the incorrect parameter is specified.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

3.2. Unsubscribe from the Call Service

Use the following API to unsubscribe from the call service:

Prototype:

```
s8 adl_callUnsubscribe ( adl_callHdlr_f CallHandler )
```

You can only unsubscribe from a subscribed call service.

Parameters:

CallHandler: This parameter is the callback function which is called whenever the Open AT Application receives an event related to the call service. For parameter information, refer to the section on the adl_callSubscribe API.

Returned Values:

OK (Numeric Value = 0) is returned on successful execution of the API.
ADL_RET_ERR_PARAM (Numeric Value = -2) is returned if the input parameter (CallHandler) is set to NULL.
ADL_RET_ERR_UNKNOWN_HDL (Numeric Value = -3) is returned if the provided handler value is incorrect.
ADL_RET_ERR_NOT_SUBSCRIBED (Numeric Value = -5) is returned if the service is not subscribed.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTE :
The call service can be subscribed to multiple times in the Open AT Application.
You can only unsubscribe from a subscribed handler of the call service, or else an error is returned

3.3. Setup an outgoing call

This API is used to setup an outgoing GSM voice/data call from a specific port.

Prototype:

s8 adl_callSetupExt (ascii * PhoneNum, u8 Mode, adl_port_e Port)

ADL also provides adl_callSetup () API which can be used for setting up a call. This function executes adl_callSetupExt on Open AT virtual port. Note that it is not possible to forward the events generated by this API to external application.

Parameters:

PhoneNum: The phone number to which the call should be made.
Mode: Mode of the call (Voice/Data)
Port: The port from which the call should be made. The values this parameter can take are defined in the adl_port_e enum.

Returned Values:

OK is returned on success
ADL_RET_ERR_PARAM is returned if incorrect parameter is mentioned.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTE :
As adl_callSetup() API does not have any parameter to mention the port. This API makes an outgoing call from Open AT OS virtual port.

3.4. Hangup a call

This API is used to hang-up an outgoing/incoming GSM voice/data call from a specific port.

Prototype:

s8 adl_callHangupExt (adl_port_e Port)

ADL also provides adl_callHangup () API which can be used to hang-up a call. This function executes adl_callHangupExt on Open AT virtual port. Note that it is not possible to forward the events generated by this API to external application.

Parameters:

Port: The port from which the call should be disconnected. The values this parameter can take are defined in the adl_port_e enum.

Returned Values:

OK is returned on success
ADL_RET_ERR_PARAM is returned if incorrect parameter is mentioned.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTE :
As adl_callHangup() API is also provided by Open AT OS which does not have any parameter to mention the port. This API does the hang-up from Open AT OS virtual port.

3.5. Answer a call

This API is used to answer an incoming GSM voice/data call from a specific port.

Prototype:

s8 adl_callAnswerExt (adl_port_e Port)

ADL also provides adl_callAnswer () API which can be used to answer an incoming call. This function executes adl_callAnswerExt on Open AT virtual port. Note that it is not possible to forward the events generated by this API to external application.

Parameters:

Port: The port from which the call should be answered. The values this parameter can take are defined in the adl_port_e enum.

Returned Values:

OK is returned on success
ADL_RET_ERR_PARAM is returned if incorrect parameter is mentioned.
ADL_RET_ERR_SERVICE_LOCKED is returned if it is called from a low level interrupt handler.

NOTE :
As adl_callAnswer() API does not have any parameter to mention the port. This API accepts call from Open AT OS virtual port.

4. Sample Code

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

/* This function is called whenever any call event is received */
s8 CallHandler (   u16 Event, u32 Call_ID)
{
  ascii RspStr [ 200 ];
  s8 retValue = ADL_CALL_FORWARD;
  /* Switch on event type */
  wm_sprintf(RspStr, "\r\n Event %d received\r\n", Event);
  adl_atSendResponse (ADL_AT_RSP,RspStr);
  switch (Event)
  {
    case ADL_CALL_EVENT_RING_VOICE:
      TRACE((1,"ADL_CALL_EVENT_RING_VOICE"));
      break;
    case ADL_CALL_EVENT_RING_DATA:
      TRACE((1,"ADL_CALL_EVENT_RING_DATA"));
      break;
    case ADL_CALL_EVENT_NEW_ID:
      TRACE((1,"ADL_CALL_EVENT_NEW_ID"));
      break;
    case ADL_CALL_EVENT_RELEASE_ID:
      TRACE((1,"ADL_CALL_EVENT_RELEASE_ID"));
      break;
    case ADL_CALL_EVENT_ALERTING:
      TRACE((1,"ADL_CALL_EVENT_ALERTING"));
      break;
    case ADL_CALL_EVENT_NO_CARRIER:
      TRACE((1,"ADL_CALL_EVENT_NO_CARRIER"));
      break;
    case ADL_CALL_EVENT_NO_ANSWER:
      TRACE((1,"ADL_CALL_EVENT_NO_ANSWER"));
      break;
   case ADL_CALL_EVENT_BUSY:
      TRACE((1,"ADL_CALL_EVENT_BUSY"));
      break;
    case ADL_CALL_EVENT_SETUP_OK:
      TRACE((1,"ADL_CALL_EVENT_SETUP_OK"));
      adl_callHangup ( );
      retValue = ADL_CALL_NO_FORWARD_ATH;
      break;
    case ADL_CALL_EVENT_ANSWER_OK:
      TRACE((1,"ADL_CALL_EVENT_ANSWER_OK"));
      break;
    case ADL_CALL_EVENT_HANGUP_OK:
      TRACE((1,"ADL_CALL_EVENT_HANGUP_OK"));
      adl_callUnsubscribe (CallHandler);
      break;
    case ADL_CALL_EVENT_SETUP_OK_FROM_EXT:
      TRACE((1,"ADL_CALL_EVENT_SETUP_OK_FROM_EXT"));
      break;
    case ADL_CALL_EVENT_ANSWER_OK_FROM_EXT:
      TRACE((1,"ADL_CALL_EVENT_ANSWER_OK_FROM_EXT"));
      break;
    case ADL_CALL_EVENT_HANGUP_OK_FROM_EXT:
      TRACE((1,"ADL_CALL_EVENT_HANGUP_OK_FROM_EXT"));
      break;
    case ADL_CALL_EVENT_AUDIO_OPENNED:
      TRACE((1,"ADL_CALL_EVENT_AUDIO_OPENNED"));
      break;
    case ADL_CALL_EVENT_ANSWER_OK_AUTO:
      TRACE((1,"ADL_CALL_EVENT_ANSWER_OK_AUTO"));
      break;
    case ADL_CALL_EVENT_RING_GPRS:
      TRACE((1,"ADL_CALL_EVENT_RING_GPRS"));
      break;
    case ADL_CALL_EVENT_SETUP_FROM_EXT:
      TRACE((1,"ADL_CALL_EVENT_SETUP_FROM_EXT"));
      break;
    case ADL_CALL_EVENT_SETUP_ERROR_NO_SIM:
      TRACE((1,"ADL_CALL_EVENT_SETUP_ERROR_NO_SIM"));
      break;
    case ADL_CALL_EVENT_SETUP_ERROR_PIN_NOT_READY:
      TRACE((1,"ADL_CALL_EVENT_SETUP_ERROR_PIN_NOT_READY"));
      break;
    case ADL_CALL_EVENT_SETUP_ERROR:
      TRACE((1,"ADL_CALL_EVENT_SETUP_ERROR"));
      break;
    default:
      break;
  }
  return retValue;
}

bool wind_4_handler(adl_atUnsolicited_t * paras)
{
  TRACE((1,"Inside wind 4 handler"));
```

```
   adl_callSetup ("9844472947", ADL_CALL_MODE_VOICE);
   return (0);
 }

/* Open AT entry point function */
void adl_main (adl_InitType_e adlInitType)
{
   s8 retValue;
   /* Subscribe for call service */
   /* Handler is CallHandler */
   retValue = adl_callSubscribe (CallHandler);
   if (retValue >= 0)
   {
     /* subscribe to wind4 unsolicited response in order to wait for wind4
        response to occur*/
     adl_atUnSoSubscribe("+WIND: 4",wind_4_handler);
   }
}
```

Summary

The following points have been covered in this chapter

- **Open AT OS provides the APIs to handle events related to call.**
- **Events related to the call service are issued to the Open AT Application by the external application and the embedded module.**
- **To subscribe to a call service, use the adl_callSubscribe function.**
- **To unsubscribe from a call service, use the adl_callUnsubscribe function.**
- **To set up a call, use the adl_callSetup function.**
- **To hang up a call, use the adl_callHangup function.**
- **To answer a call, use the adl_callAnswer function.**

CHAPTER 24

General Packet Radio Service

1. Objective

This chapter introduces the General Packet Radio Service (GPRS) and its usage in embedded module. It covers in detail the APIs and events related to GPRS, and also include prototypes to help you work with this service. This information is provided through the following sections of this chapter:

- Overview of GPRS service
- GPRS-related events
- GPRS-related APIs

2. Overview of GPRS

GPRS is a packet-based, wireless communication service that extends the GSM mobile data system using multiple slots of the radio channel for faster transmission speeds. It does not set up a continuous channel from a portable terminal for the transmission and reception of data, but transmits and receives data in packets, thereby making efficient use of the available radio spectrum. This makes it easier and more practical to send and receive data between mobile devices.

The main benefits of GPRS are that it reserves radio resources only for situations when there is data that must be sent, and that it reduces reliance on traditional circuit-switched network elements. The mobile stations in a cell share the same radio resources. Only when a station has data to send, does it gain control of the resources. In this way there is better utilization of the radio resources. In addition to this, GPRS allows improved quality of data services in terms of reliability, response time, and features supported. Hence, GPRS is of utmost importance in migration of the existing GSM network to a 3G network.

Open AT OS provides the GPRS API set to use the GPRS functionality and to handle GPRS-related events, errors and indications.

You should subscribe to the GPRS service, set up and activate the GPRS connection in order to use the GPRS network. Then use Flow Control APIs to communicate over the GPRS channel. See section on FCM for more details.

Once a GPRS connection is established, the PDP (Packet data Protocol) is used to indicate the type of packet (IP or PPP) that you want to send over the established connection.

3. GPRS-related Events

GPRS-related events can be issued both by the network and by the mobile equipment. These events indicate the mobile or network status.
You assign a Cid (Context ID) number which is used to identify a GPRS data circuit. The attributes of the context indicate the route for the Data Access Point Name, type of data application (usually IP), quality of service and whether or not the data link is activated.

The following table lists the GPRS-related events/Cid pairs, their corresponding numeric values and descriptions:

GPRS Events/Cid	Numeric Value	Description
ADL_GPRS_EVENT_RING_GPRS	0	Incoming PDP context activation is requested by the network.
ADL_GPRS_EVENT_NW_CONTEXT_DEACT / X	1	The network has forced the deactivation of the Cid X
ADL_GPRS_EVENT_ME_CONTEXT_DEACT/ X	2	The ME has forced the deactivation of the Cid X
ADL_GPRS_EVENT_NW_DETACH	3	The network has forced the detachment of the ME.
ADL_GPRS_EVENT_ME_DETACH	4	The ME has forced a network detachment or lost the network connection.
ADL_GPRS_EVENT_NW_CLASS_B	5	The network has forced the ME on class B.
ADL_GPRS_EVENT_NW_CLASS_CG	6	The network has forced the ME on class CG.
ADL_GPRS_EVENT_NW_CLASS_CC	7	The network has forced the ME on class CC.
ADL_GPRS_EVENT_ME_CLASS_B	8	The ME has changed its class to class B.
ADL_GPRS_EVENT_ME_CLASS_CG	9	The ME has changed its class to class CG.
ADL_GPRS_EVENT_ME_CLASS_CC	10	The ME has changed its class to class CC.
ADL_GPRS_EVENT_NO_CARRIER	11	The activation of the external application with 'ATD * 999' (ppp dialing) has been disconnected.
ADL_GPRS_EVENT_DEACTIVATE_OK/ X	12	The deactivation requested with the adl_gprsDeact() function was successful on the Cid X.
ADL_GPRS_EVENT_DEACTIVATE_OK_FROM_EXT/ X	13	The deactivation requested by the external application succeeded on the Cid X
ADL_GPRS_EVENT_ANSWER_OK	14	The incoming PDP activation with adl_gprsDeact () was accepted successfully.
ADL_GPRS_EVENT_ANSWER_OK_FROM_EXT	15	The incoming PDP activation was accepted successfully by the external application.
ADL_GPRS_EVENT_ACTIVATE_OK/ X	16	The activation requested with adl_gprsDeact () on the Cid X was successful.

ADL_GPRS_EVENT_GPRS_DIAL_OK_FROM_EXT/ X	17	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	18	The activation requested by the external application on the Cid X has succeeded.
ADL_GPRS_EVENT_HANGUP_OK_FROM_EXT	19	The incoming PDP activation was successfully rejected by the external application.
ADL_GPRS_EVENT_DEACTIVATE_KO/ X	20	The deactivation requested with adl_gprsDeact () on the Cid X has failed.
ADL_GPRS_EVENT_DEACTIVATE_KO_FROM_EXT/ X	21	The deactivation requested by the external application on the Cid X has failed.
ADL_GPRS_EVENT_ACTIVATE_KO_FROM_EXT/ X	22	The activation requested by the external application on the Cid X has failed.
ADL_GPRS_EVENT_ACTIVATE_KO/ X	23	The activation requested with adl_gprsAct () on the Cid X has failed.
ADL_GPRS_EVENT_ANSWER_OK_AUTO	24	The incoming PDP context activation was automatically accepted by the ME.
ADL_GPRS_EVENT_SETUP_OK/ X	25	The setup of the Cid X with adl_gprsSetup () was successful.
ADL_GPRS_EVENT_SETUP_KO/ X	26	The setup of the Cid X with adl_gprsSetup () has failed.
ADL_GPRS_EVENT_ME_ATTACH	27	The ME has forced a network attachment.
ADL_GPRS_EVENT_ME_UNREG	28	The ME is not registered to the network.
ADL_GPRS_EVENT_ME_UNREG_SEARCHING	29	The ME is not registered, but is searching for a new operator to register with.

4. GPRS-related APIs

Following is the list of APIs that provide GPRS-related functionality:

adl_gprsSubscribe
adl_gprsUnsubscribe
adl_gprsSetup
adl_gprsSetupExt
adl_gprsAct
adl_gprsActExt
adl_gprsDeact
adl_gprsDeactExt
adl_gprsGetCidInformation

adl_gprsIsAnIPAddress

4.1. Subscribing to the GPRS Service

Use the following API to subscribe to the GPRS network. It specifies a callback function which is invoked whenever GPRS-related events are received.

Prototype:

s8 adl_gprsSubscribe (adl_gprsHdlr_f GprsHandler)

Parameters:

GprsHandler: Callback function which is called whenever a GPRS-related event is received by the Open AT Application:

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

Parameters:

The input argument of this callback function is the GPRS event and GPRS Cid. GPRS Events and Cid are explained in the preceding section – GPRS-Related Events.

If Cid X is not defined, the enum ADL_CID_NOT_EXIST will be used.

Returned Values:

The events returned by this handler can be:

Value	Description
ADL_GPRS_FORWARD	The GPRS event indications will be sent to the external application.
ADL_GPRS_NO_FORWARD	The GPRS event indications will not be sent to the external application.
ADL_GPRS_NO_FORWARD_ATH	The GPRS event indications will not be sent to the external application. The Open AT Application will terminate the incoming activation request by sending an 'ATH' command.
ADL_GPRS_NO_FORWARD_ATA	The GPRS event indications will not be sent to the external application. The Open AT Application will terminate the incoming activation request by sending an 'ATA' command.

Returned Values:

OK is returned on successful subscription for GPRS.

ADL_RET_ERR_PARAM is returned if some parameter is invalid.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

4.2. Unsubscribing from the GPRS Service

Use the following API to unsubscribe from the GPRS service. After executing this function, the callback function (Gprshandler) will not be called anymore when any GPRS events are received.

Prototype:

```
s8 adl_gprsUnsubscribe (adl_gprsHdlr_f Gprshandler)
```

Parameters:

Gprshandler: Handler function used with the adl_gprsSubscribe API.

Returned Values:

This API returns the following values:
OK: This value is returned on successful execution of the API.
ADL_RET_ERR_PARAM: This value is returned if the given parameter is incorrect.
ADL_RET_ERR_UNKNOWN_HDL: This value is returned if the provided handler is unknown.
ADL_RET_ERR_NOT_SUBSCRIBED: This value is returned if the service is not subscribed.
ADL_RET_ERR_BAD_STATE: The service is currently processing another GPRS API. The Open AT Application should wait for the corresponding event (indication of the end of processing) in the GPRS handler.
ADL_RET_ERR_SERVICE_LOCKED: if the function is called from a low level interrupt handler.
You can only unsubscribe from a subscribed GPRS service. If you try to unsubscribe from GPRS (using adl_gprsUnsubscribe) without having subscribed to it, the ADL_RET_ERR_NOT_SUBSCRIBED error will be returned.

4.3. PDP Context Information

The following API should be used to get information about a specific activated PDP context (identified by its Cid):

Prototype:

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

You can also use this API to get information about the DNS (domain name system) and gateway IP address.

Parameters:

Cid: The Cid of the PDP context about which we want information.
Infos: This parameter is a structure containing the information of the activated PDP context. The structured is as follows:

```
typedef struct
{
   u32 LocalIP;
   u32 DNS1;
   u32 DNS2;
   u32 Gateway;
}adl_gprsInfosCid_t;
```

Parameters:

LocalIP: Local IP address of the mobile station (only if is activated, or else it is 0).
DNS1: First DNS IP address (only if it is activated, or else it is 0).
DNS2: Second DNS IP address (only if it is activated, or else it is 0).
Gateway: Gateway IP address (only if it is activated, or else it is 0).

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned in case of a invalid parameter. Note that the Cid value must be in the range of 1 to 4.
ADL_RET_ERR_BAD_STATE is returned when the service is still processing another GPRS API. The Open AT Application should wait for the corresponding event (indication of the end of processing) in the GPRS handler.
ADL_RET_ERR_PIN_KO is returned if PIN is not entered, or "+WIND:4" indication has not yet occurred.
ADL_NO_GPRS_SERVICE is returned if GPRS service is not supported by the product.

NOTE : If your Open AT Application opens the GPRS FCM flow in addition to subscribing to GPRS services, then the adl_gprsAct () API must be called before opening the flow. Similarly adl_gprsDeact () API must be called after receiving the ADL_FCM_EVENT_FLOW_CLOSED event.
Give due attention to the return value of the gprshandler() function (specified in adl_gprsSubscribe () API). Your external application will receive GPRS-related events based on the return values of gprshandler() function..

4.4. Configuring the APN parameters and setting up GPRS Service

This API is used to setup the PDP context id. Open AT OS allows to setup 4 context ids at a time. The context id contains the following information:

- Access point name
- User name
- Password
- Static IP address if any

Prototype:

```
s8 adl_gprsSetupExt ( u8 Cid, adl_gprsSetupParams_t param, adl_port_e Port )
```

ADL also provides adl_gprsSetup () API which can be used to set up a GPRS service. This function executes adl_gprsSetupExt on Open AT virtual port. Note that it is not possible to forward the events generated by this API to external application.

Parameters:

Cid: Context identifier of the PDP context. This parameter can take values from 1 to 4.
param: This parameter contains the information related to context id. This includes the APN, user name, password and IP address. This parameter is defined by a structure adl_gprsSetupParams_t which is defined below:

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

Port: The port from which the setup should be made. The values this parameter can take are defined in the adl_port_e enum.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if incorrect parameter is entered.
ADL_RET_ERR_PIN_KO is returned if PIN is not entered.
ADL_RET_ERR_BAD_STATE is returned if GPRS service is processing another GPRS related API.
ADL_GPRS_CID_NOT_DEFINED is returned if context id is already activated.
ADL_GPRS_NO_SERVICE is returned if GPRS not supported.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTE :
1. As adl_gprsSetup() API does not have any parameter to mention the port. This API defines the context id for Open AT OS virtual port.
2. Multiple context ids (up to 4) can be defined but only one can be activated at a time.

4.5. The adl_gprsActExt () API

This API is used to activate a previously defined context id from a specific port. After activation, network allocates an IP address to the embedded module.

Prototype:

s8 adl_gprsActExt (u8 Cid, adl_port_e Port)

ADL also provides adl_gprsAct () API which can be used to activate a previously defined context id. This function executes adl_gprsActExt on Open AT virtual port. Note that it is not possible to forward the events generated by this API to external application.

Parameters:

Cid: Context identifier of the PDP context. This parameter can take values from 1 to 4.
Port: The port from which the context id should be activated. The values this parameter can take are defined in the adl_port_e.

Returned Values:

OK on success
ADL_RET_ERR_PARAM, if incorrect parameter is entered.
ADL_RET_ERR_PIN_KO, if PIN is not entered
ADL_RET_ERR_BAD_STATE, if GPRS service is not available.
ADL_GPRS_CID_NOT_DEFINED, if CID is already activated
ADL_GPRS_NO_SERVICE, if GPRS not supported.
ADL_RET_ERR_SERVICE_LOCKED: if the function is called from a low level interruption handler.

NOTE :
As adl_gprsAct () API does not have any parameter to mention the port. This API does the activation from Open AT OS virtual port.

4.6. The adl_gprsDeactExt () API

This API is used to deactivate previously activated context id for a specific port.

Prototype:

s8 adl_gprsDeactExt (u8 Cid, adl_port_e Port)

ADL also provides adl_gprsDeact () API which can be used to deactivate a previously activated context id. This function executes adl_gprsDeactExt on Open AT virtual port. Note that it is not possible to forward the events generated by this API to external application.

Parameters:

Cid: Context identifier of the PDP context. This parameter can take values from 1 to 4.
Port: The port from which the context id should be deactivated. The values this parameter can take are defined in the adl_port_e enum.

Returned Values:

OK is returned on success.

ADL_RET_ERR_PARAM is returned if incorrect parameter is entered.
ADL_RET_ERR_PIN_KO is returned if PIN is not entered.
ADL_RET_ERR_BAD_STATE is returned if GPRS service is not available.
ADL_GPRS_CID_NOT_DEFINED is returned if context id is already activated.
ADL_GPRS_NO_SERVICE is returned if GPRS not supported.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interruption handler.

NOTE :
As adl_gprsDeactExt() API does not have any parameter to mention the port, this API deactivates the context id from Open AT OS virtual port.

4.7. The adl_gprsIsAnIPAddress () API

This API is used to check whether provided string is a valid IP address. The valid address is based on the "a.b.c.d" format where a, b, c and d are integer values between 0 and 255.

Prototype:

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

Parameters:

AddressStr: The string to check whether it is a valid IP address.

Returned Values:

TRUE is returned if the string is a valid IP address.
FALSE is returned if the string is not a valid IP address.

5. Sample Code

```
/* sample code implementing the GPRS functionality */
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

/* Global variables */
adl_gprsSetupParams_t GPRS_Params;
u8 Cid;
ascii APN[] = "airtelgprs.com";
ascii UserID[] = "airtel";
ascii PWD[] = "tango";

s8 GPRS_Handler ( u16 Event,u8 Cid )
{
  s8 ret;
  TRACE (( 1, "GPRS Handler: Event received = %d", Event ));
  switch ( Event )
```

```
            {
             case ADL_GPRS_EVENT_ME_CONTEXT_DEACT:
              TRACE (( 1, "ADL_GPRS_EVENT_ME_CONTEXT_DEACT" ));
              break;
             case ADL_GPRS_EVENT_NW_DETACH:
              TRACE (( 1, "ADL_GPRS_EVENT_NW_DETACH" ));
              break;
             case ADL_GPRS_EVENT_ME_DETACH:
              TRACE (( 1, "ADL_GPRS_EVENT_ME_DETACH" ));
              break;
             case ADL_GPRS_EVENT_NO_CARRIER:
              TRACE (( 1, "ADL_GPRS_EVENT_NO_CARRIER" ));
              break;
             case ADL_GPRS_EVENT_DEACTIVATE_OK:
              TRACE (( 1, "ADL_GPRS_EVENT_DEACTIVATE_OK" ));
              break;
             case ADL_GPRS_EVENT_ACTIVATE_OK:
              TRACE (( 1, "GPRS Event activated"));
              break;
             case ADL_GPRS_EVENT_DEACTIVATE_KO:
              TRACE (( 1, "ADL_GPRS_EVENT_DEACTIVATE_KO" ));
              break;
             case ADL_GPRS_EVENT_ACTIVATE_KO:
              TRACE (( 1, "ADL_GPRS_EVENT_ACTIVATE_KO" ));
              break;
             case ADL_GPRS_EVENT_SETUP_OK:
              TRACE (( 1, "ADL_GPRS_EVENT_SETUP_OK" ));
              /* Activate context */
              ret = adl_gprsAct ( Cid );
              TRACE((1,"Return value of adl_gprsAct = %d", ret));
              break;
             case ADL_GPRS_EVENT_SETUP_KO:
              TRACE (( 1, "ADL_GPRS_EVENT_SETUP_KO" ));
              break;
             case ADL_GPRS_EVENT_ME_ATTACH:
              TRACE (( 1, "ADL_GPRS_EVENT_ME_ATTACH" ));
              break;
             case ADL_GPRS_EVENT_ME_UNREG:
              TRACE (( 1, "ADL_GPRS_EVENT_ME_UNREG" ));
              ret = adl_gprsSetup (Cid, GPRS_Params);
              TRACE((1,"Return value of adl_gprsSetup = %d", ret));
              break;
             case ADL_GPRS_EVENT_ME_UNREG_SEARCHING:
              TRACE (( 1, "ADL_GPRS_EVENT_ME_UNREG_SEARCHING" ));
              ret = adl_gprsSetup (Cid, GPRS_Params);
              TRACE((1,"Return value of adl_gprsSetup = %d", ret));
              break;
```

```
        default:
          TRACE (( 1, "Embedded: Default section called."));
          break;
      }
      return ADL_GPRS_FORWARD;
    }

    bool wind4Handler(adl_atUnsolicited_t *params)
    {
      s8 ret;
      TRACE (( 1, "WIND 4 handler" ));
      Cid = 1;
      GPRS_Params.APN = APN;
      GPRS_Params.Login = UserID;
      GPRS_Params.Password = PWD;
      ret = adl_gprsSubscribe( GPRS_Handler);
      TRACE((1,"Return value of adl_gprsSubscribe = %d", ret));
      ret = adl_gprsSetup( Cid, GPRS_Params );
      TRACE((1,"Return value of adl_gprsSetup = %d", ret));
      return TRUE;
    }

    void adl_main ( adl_InitType_e InitType )
    {
      TRACE (( 1, "Embedded Application: Main" ));
      adl_atUnSoSubscribe ( "+WIND: 4", wind4Handler );
    }
```

Summary

The following points have been covered in this chapter

- **GPRS (General packet radio service) allows information to be sent and received across mobile phone networks.**
- **To subscribe to the GPRS service use the adl_gprsSubscribe () API.**
- **To set up the PDP context parameters for GPRS network connection, use the adl_gprsSetup () API. The parameters required to connect to the GPRS network are APN (access point name), log in ID of the GPRS account and the password of the GPRS account.**
- **To activate the PDP context, use adl_gprsAct () API. This API should be used before the FCM flow is opened.**
- **To deactivate the PDP context, use the adl_gprsDeact () API. This API should be used only when the FCM flow is closed - that is, after receiving the ADL_FCM_EVENT_FLOW_CLOSED event to avoid a embedded module lock.**
- **To unsubscribe from the GPRS service, use the adl_gprsUnsubscribe () API.**

CHAPTER 25

Application Safe Mode Service

1. Objective

This chapter introduces the application safe mode service and its related APIs. The various events related to the application safe mode are explained here. The aim of this chapter is to help you understand ways to handle the +WDWL and +WOPEN command issued by external application to embedded module. The following points are covered in this chapter:

- Overview of application safe mode service
- Events related to the application safe mode
- APIs related to the application safe mode service

APIs related to the application safe mode service are declared in the adl_safe.h header file.

2. Overview of the Application Safe Mode Service

Open AT OS provides the application safe mode service to handle the +WDWL and +WOPEN commands.
Consider a case when the embedded Open AT Application is running on the embedded module and you do not want the embedded Open AT Application to be terminated by any external application event. At this time if any external application issues the AT+WOPEN=0 command to the embedded module to terminate the Open AT Application, the

Open AT Application stops running. In such situations, use the application safe mode service to capture the command and prevent it from being processed on the Firmware.

Another example could be a case when an external application issues the +WDWL command to overwrite the currently running Open AT Application. Use the application safe mode service to manage these types of situations.

3. Events Related to the Application Safe Mode Service

The +WOPEN and +WDWL commands received by the Open AT Application are events related to the application safe mode, and are issued by the external application to the embedded module.

Following is a list of application safe mode events and their corresponding numeric values:

Safe Mode Events	Description
ADL_SAFE_CMD_WDWL	This event is received when the AT+WDWL command is issued by an external application.
ADL_SAFE_CMD_WDWL_READ	This event is received when the AT+WDWL? command is issued by an external application.
ADL_SAFE_CMD_WDWL_OTHER	This event is received when the custom +WDWL command syntax is issued by the external application.
ADL_SAFE_CMD_WOPEN_STOP	This event is received when the AT+WOPEN=0 command is issued by the external application.
ADL_SAFE_CMD_WOPEN_START	This event is received when the AT+WOPEN=1 command is issued by the external application
ADL_SAFE_CMD_WOPEN_GET_VERSION	This event is received when the AT+WOPEN=2 command is issued by the external application
ADL_SAFE_CMD_WOPEN_ERASE_OBJ	This event is received when the AT+WOPEN=3 command is issued by the external application
ADL_SAFE_CMD_WOPEN_ERASE_APP	This event is received when the AT+WOPEN=4 command is issued by the external application
ADL_SAFE_CMD_WOPEN_SUSPEND_APP	This event is received when AT+WOPEN=5 command is issued by external application.
ADL_SAFE_CMD_WOPEN_READ	This event is received when the AT+WOPEN? command is issued by the external application.
ADL_SAFE_CMD_WOPEN_TEST	This event is received when the AT+WOPEN=? command is issued by the external application.
ADL_SAFE_CMD_WOPEN_OTHER	This event is received when the custom +WOPEN command syntax is issued by the external application.
ADL_SAFE_CMD_WOPEN_AD_GET_SIZE	This event is received when the AT+WOPEN=6 command is issued by the external application for retreiving the A&D memory size.
ADL_SAFE_CMD_WOPEN_AD_SET_SIZE	This event is received when the AT+WOPEN=6,<size> command is issued by the external application for setting the A&D memory size.

4. Application Safe Mode Service APIs

Following is the list of APIs which are used to handle the events related to the application safe mode:

- adl_safeSubscribe
- adl_safeUnsubscribe
- adl_safeRunCommand

adl_safe.h is the header file which is required to use the Application Safe Mode service.

4.1. Subscribing to Application Safe Mode Service

The following API should be used to subscribe to the application safe mode service:

Prototype:

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

This API subscribes to application safe mode service and specifies a callback function which is called whenever events related to the application safe mode are received.

Parameters:

WDWLopt: Firmware provides the AT+WDWL? command (Read mode) to read the status of the software version. WDWLopt is the additional option for the WDWL command. For example, if you pass a value of ADL_CMD_TYPE_PARA|0x11 , then the AT+WDWL command can be executed with exactly one argument. Refer to the *Subscribing to AT commands* section for more details.

WOPENopt: Firmware provides the AT+WOPEN? (Read mode) command, the AT+WOPEN=? (Test mode) and the AT+WOPEN=<param> (Param Mode) command to erase, run or stop the Open AT Application. WOPENopt is the additional option for the WOPEN command. For example, if you pass the value of 0x22 , then the AT+WDWL command can be executed with up to two arguments. Refer to the Subscribing to AT commands section for more details.

SafeHandler: The callback function which is called whenever an event related to the application safe mode is received by the Open AT Application. The Handler is defined as the following function pointer:

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

Parameters:

CmdType: The CmdType argument of this callback function is an event related to the application safe mode. These events are explained in the section on Safe Mode Events.
Paras: The paras argument of this callback function is the structure which holds the command and its parameter to which you have subscribed. Refer to the Subscribing to AT commands section for more detail.

Return Value:

FALSE: If the return value of this handler is FALSE then the command is not forwarded to Firmware. In this case command is processed by Open AT Application. If the return value of this

TRUE: If the return value of this handler is TRUE then the command is forwarded to lower layer forprocessing, which sends the response to the external application.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if the parameter is given an incorrect value.
ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the service is already subscribed.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

4.2. Unsubscribing from the Application Safe Mode Service

The following API should be used to unsubscribe from the application safe mode service:

Prototype:

```
s8 adl_safeUnsubscribe (adl_safeHdlr_f SafeHandler)
```

It should be noted that you can only unsubscribe from a subscribed application safe mode service.

Parameters:

SafeHandler: This parameter is the callback function which is called whenever an event related to the application safe mode is received by the Open AT Application. For parameter information, refer to the section on the
s8 adl_safeSubscribe API

Returned Values:

This API returns the following value:
OK is returned on success.
ADL_RET_ERR_PARAM is returned if the parameter is given an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handler value is incorrect.
ADL_RET_ERR_NOT_SUBSCRIBED is returned if the service is not subscribed.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

4.3. adl_safeRunCommand function

This function is used to execute the +WDWL and +WOPEN commands provided by Firmware. This function cannot be used to execute custom +WDWL and +WOPEN command. This API is declared as follows:

Prototype:

```
s8 adl_safeRunCommand ( adl_safeCmdType_e CmdType, adl_atRspHandler_t rsphdl)
```

Parameters:

CmdType: The CmdType argument of this function can have any event related to the application safe mode service other than ADL_SAFE_CMD_WDWL_OTHER and ADL_SAFE_CMD_WOPEN_OTHER.
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 pin.

rsphdl: The rsphdl argument is the callback function which is called whenever a response is received by Open AT Application for the +WDWL and +WOPEN commands. If its value is set to NULL then all the responses are forwarded to the external application. Refer to argument RspHdl of API adl_atCmdCreate for more details on this argument.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if the parameter is given an incorrect value.
LOCKED is returned if the function is called from a low level interrupt handler.

NOTE :
The application safe mode service can be subscribed only once in the Open AT Application.
+WDWL command is at least subscribed in ACTION and READ mode for the API adl_safeSubscribe.
+WOPEN command is at least subscribed in TEST, READ and PARAM mode for the API adl_safeSubscribe.
You can only unsubscribe from a subscribed handler of the application safe mode service, or else an error is returned.
ADL_SAFE_CMD_WDWL_OTHER and ADL_SAFE_CMD_WOPEN_OTHER values are not allowed for the argument CmdType of the adl_safeRunCommand function.
By default all the responses are subscribed for the rsphdl argument of adl_safeRunCommand function

5. Sample Code

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;

/* This function is called whenever any SIM event is received */
bool MySafeHandler (   adl_safeCmdType_e CmdType,
adl_atCmdPreParser_t * paras)
{
  bool bReturn = TRUE;
  /* Switch on event type */
  switch (CmdType)
  {
    case ADL_SAFE_CMD_WDWL_READ:
    case ADL_SAFE_CMD_WOPEN_START:
    case ADL_SAFE_CMD_WOPEN_GET_VERSION:
    case ADL_SAFE_CMD_WOPEN_ERASE_OBJ:
    case ADL_SAFE_CMD_WOPEN_ERASE_APP:
    case ADL_SAFE_CMD_WOPEN_READ:
    case ADL_SAFE_CMD_WOPEN_TEST:
      adl_atSendResponse (ADL_AT_RSP,"\r\nStandard Command allowed\r\n");
      break;
    case ADL_SAFE_CMD_WDWL:
    case ADL_SAFE_CMD_WOPEN_STOP:
    case ADL_SAFE_CMD_WOPEN_SUSPEND_APP:
      adl_atSendResponse (ADL_AT_RSP,"\r\nDownload or erase is not allowed\r\n");
      bReturn = FALSE;
      break;
    case ADL_SAFE_CMD_WDWL_OTHER:
    case ADL_SAFE_CMD_WOPEN_OTHER:
      adl_atSendResponse (ADL_AT_RSP,"\r\nCustom command is received\r\n");
      /* If custom command is received then run AT+WOPEN=? command */
      /* Handler is set to NULL - Responses will be forwarded to external application */
      adl_safeRunCommand (ADL_SAFE_CMD_WOPEN_TEST, NULL);
      /* Un Subscribe from application safe mode service */
      adl_safeUnsubscribe (MySafeHandler);
      break;
  }
  return bReturn;
}

/* Open AT entry point function */
void adl_main (adl_InitType_e adlInitType)
{

  /* Subscribe for application safe mode service */
  /* Handler is MySafeHandler */
  /* WDWL option: 1 parameter and default Action and Read Mode */
  /* WOPEN option: 2 parameters and default Read, Test and one Param mode */
  adl_safeSubscribe (ADL_CMD_TYPE_PARA | 0x11, 0x22, MySafeHandler);
}
```

Summary

The following points have been covered in this chapter

- **Open AT OS provides the application safe mode service to handle events related to this mode.**
- **Events related to the application safe mode are issued by the external application to the embedded module.**
- **To subscribe an application safe mode service, use the adl_safeSubscribe function.**
- **To unsubscribe from an application safe mode service, use the adl_safeUnSubscribe function.**
- **To run the standard WDWL and WOPEN command use the adl_safeRunCommand function**

CHAPTER 26

Open SIM Access (OSA) Service

1. Objective

The embedded module provides interface to connect multiple SIM cards. The embedded Module has a defined default interface for a SIM card. Along with this, user can connect SIM cards through different hardware interfaces such as UART, I2C, SPI, USB etc.
When additional SIM card is connected, the OSA service can be used to select the additional SIM card.
The SIM driver in Firmware programs SIM card using set of packets/commands. These packets/commands are referred as **APDU** and are defined in 3GPP TS 11.11.

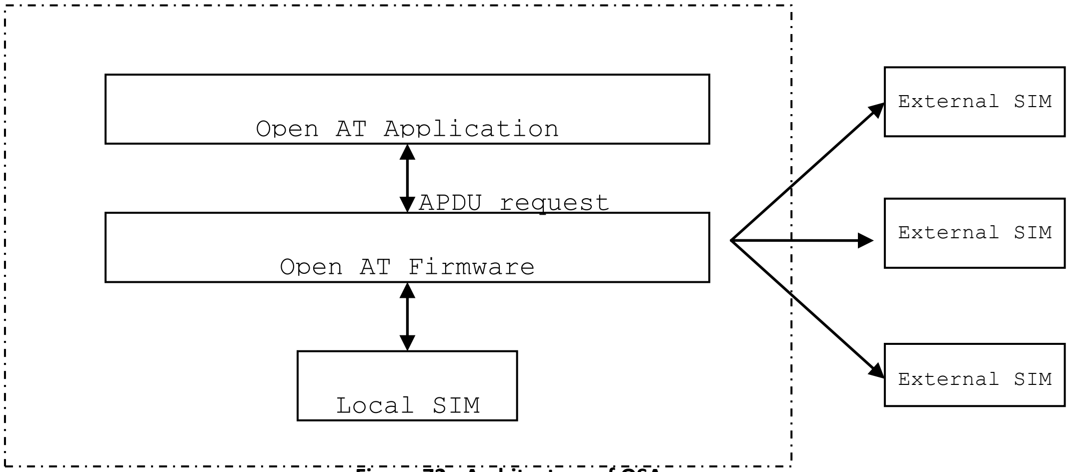


Figure 72 - Architecture of OSA

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, and I2C).
Open SIM Access feature has to be enabled on the embedded module in order to make this service available. The Open SIM Access feature state can be read using AT+WCFM=5 command: this feature state is represented by the bit 5 (00000020 in hexadecimal format) of the command response.

2. Required header file

The header file required in order to use OSA service
adl_osa.h

3. OSA Service APIs

3.1. The adl_osaSubscribe() API

This API will subscribe to the OSA service. On subscribing to the OSA service the embedded module will disconnect the local SIM and will send all the SIM requests to Open AT Application. All the SIM messages will be notified to the handler provided at the time of the subscription.

Prototype:

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

Parameter:

OsaHandler: The handler to receive all the SIM messages.

Prototype:

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

Parameter:

Event: Following are the events for OSA service:

OSA Service Events	Description
ADL_OSA_EVENT_INIT_SUCCESS	OSA service has been successfully subscribed.
ADL_OSA_EVENT_INIT_FAILURE	OSA service subscription failure.
ADL_OSA_EVENT_ATR_REQUEST	This event is notified after the successful subscription of OSA service. On receiving the ATR request the embedded module must reset the remote SIM.
ADL_OSA_EVENT_APDU_REQUEST	This event is received by the application each time embedded module has to send a APDU request to the remote SIM. It has to be forwarded to the remote SIM by the application.
ADL_OSA_EVENT_SIM_ERROR	This event is notified to the application in case an error has occurred in embedded module due to response. OSA is automatically unsubscribed on receiving error.

OSA Service Events	Description
ADL_OSA_EVENT_CLOSED	This event is received on unsubscribing to OSA service

Param: Parameter corresponding to different event received. The structure defines the various parameter that are received

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

ErrorEvent: Defines the error cause in case of event ADL_OSA_EVENT_SIM_ERROR.

Length: Length of the data received.

Data: Data corresponding to the event. The data is received in case of event ADL_OSA_EVENT_APDU_REQUEST.

Returned Value:

A positive or null handle is returned on success.
A negative value is returned on failure.
ADL_RET_ERR_PARAM is returned if there is an error in the parameter
ADL_RET_ERR_NOT_SUPPORTED is returned if the OSA feature is not enabled in embedded module
ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the service was already subscribed
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler

3.2. The adl_osaSendResponse() API

This API allows the application to post ATR and APDU request to the embedded module after receiving ADL_OSA_EVENT_ATR_REQUEST or ADL_OSA_EVENT_ APDU_REQUEST.

Prototype:

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

Parameter:

OsaHandle: The handle returned at the time of subscription of the OSA service.
Status: Status to be supplies to the embedded module in response to ATR and APDU request. Following status values are defined.
ADL_OSA_STATUS_OK: Response data buffer has been received from SIM card.
ADL_OSA_STATUS_CARD_NOT_ACCESSIBLE: No response from SIM card.
ADL_OSA_STATUS_CARD_REMOVED: SIM card has been removed.
ADL_OSA_STATUS_CARD_UNKNOWN_ERROR: Generic error code for all other errors.
Length: ATR and APDU request response buffer length.
Data: ATR and APDU buffer address.

Returned Value:

OK is returned on success.
A negative value is returned on failure.
ADL_RET_ERR_PARAM: if there is an error in the parameter
ADL_RET_ERR_UNKNOWN_HDL: if the unknown OSA handle is specified
ADL_RET_ERR_BAD_STATE: if the OSA service is not waiting for an APDU or ATR request response

3.3. The adl_osaUnsubscribe() API

This API is provided in order to unsubscribe from the OSA service.

Prototype:

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

Parameter:

OsaHandle: The handle returned at the time of subscription of the OSA service.

Returned Value:

OK is returned on success. An ADL_OSA_EVENT_CLOSED confirmation event will be received in the service handler.
A negative value is returned on failure.
ADL_RET_ERR_UNKNOWN_HDL: if the unknown OSA handle is specified
ADL_RET_ERR_SERVICE_LOCKED: if the function is called from a low level interrupt handler
ADL_RET_ERR_NOT_SUBSCRIBED: if OSA service is not subscribed using adl_osaSubscribe () API.
ADL_RET_ERR_BAD_STATE: if Firmware is waiting for an ATR or APDU request from the simcard, and unsubscription is forbidden until the simcard's request is granted.

4. Sample Code

```
// Example showing OSA service API usage.
#include "adl_global.h"
#include "adl_osa.h"
```

```
const u16 wm_apmCustomStackSize = 1024

static s32  OsaHandle;   // Handle for the OSA service
static u16 Flow = ADL_FCM_FLOW_V24_UART2;

/*This buffer will be constructed in the application according to specification */
u8 *FrameData;
u32 FrameLenght;
void MyOsaHandler ( adl_osaEvent_e Event, adl_osaEventParam_u * Param )
{
   TRACE (( 1, "MyOsaHandler: Event %d" , Event ));
   // Switch on the event type
   switch ( Event )
   {
      case ADL_OSA_EVENT_INIT_SUCCESS:
         TRACE (( 1, "MyOsaHandler: ADL_OSA_EVENT_INIT_SUCCESS" ));
         break;
     case ADL_OSA_EVENT_CLOSED:
         TRACE (( 1, "MyOsaHandler: ADL_OSA_EVENT_CLOSED" ));
         break;
     case ADL_OSA_EVENT_INIT_FAILURE:
         TRACE (( 1, "MyOsaHandler: ADL_OSA_EVENT_INIT_FAILURE" ));
         FlowFCMclose ();
         break;
     case ADL_OSA_EVENT_ATR_REQUEST:
         break;
     case ADL_OSA_EVENT_APDU_REQUEST:
         adl_osaSendResponse(OsaHandle,ADL_OSA_STATUS_OK,);
         // Send APDU frame to the remote SIM on UART 2.
         adl_fcmSend(FlowHandle,FrameData,FrameLenght);
         break;
     case ADL_OSA_EVENT_SIM_ERROR:
         {
            TRACE (( 1, "ErrorEvent : %d", Param->ErrorEvent ));
            switch ( Param->ErrorEvent )
            {
               case ADL_OSA_STATUS_OK:
                  TRACE (( 1, "ErrorEvent: ADL_OSA_STATUS_OK" ));
                  break;
               case ADL_OSA_STATUS_CARD_NOT_ACCESSIBLE:
                  TRACE (( 1, "ErrorEvent: ADL_OSA_STATUS_CARD_NOT_ACCESSIBLE"
                  ));
                  break;
               case ADL_OSA_STATUS_CARD_REMOVED:
                  TRACE (( 1, "ErrorEvent: ADL_OSA_STATUS_CARD_REMOVED" ));
                  break;
               case ADL_OSA_STATUS_CARD_UNKNOWN_ERROR:
                    TRACE (( 1, "ErrorEvent: ADL_OSA_STATUS_CARD_UNKNOWN_ERROR"
                    ));
                    break;
                }
            }
            break;
    }
}
void FlowCtrlHandler()
{
...
}
bool FlowDataHandler(u16 Length, u8 * Data)
{
    .....
                // Send the APDU and ATR response to the remote SIM conneted throught UART 2
                adl_osaSendResponse(OsaHandle,ADL_OSA_STATUS_OK,Lenght, Data);
}

void adl_main ( adl_InitType_e InitType )
{
                OsaHandle = adl_osaSubscribe ( MyOsaHandler );
    FlowHandle  = adl_fcmSubscribe ( Flow, FlowCtrlHandler, FlowDataHandler )

}
```

CHAPTER 27

List Management

1. Objective

Open AT implements its own linked list data structures using Open AT OS list management APIs. The list management functionality is extensively used in Open AT Application which needs to store data dynamically at runtime. The data type (integers/strings) of the stored data can vary depending on the type of Open AT Application and its requirements.
This chapter describes the list management API and its usage in embedded module.

2. Open AT List APIs

List APIs store data in volatile memory. Unlike the data stored using flash object IDs, data stored in the list is lost once an Open AT Application is stopped. Apart from the facility to sort and store data, you can avoid data redundancy caused by duplicate data, by enabling an attribute in the API. This ensures that only unique data are stored in the list.

3. Implementing the List Management Functionality in the embedded module

Open AT ADL library uses the basic Open AT OS list management APIs defined in the basic interface of application development. The required header for list management API wm_list.h, is included in the global ADL header file adl_global.h.
The list pointer corresponding to the list created using list management APIs is of the following type:

```
typedef void * wm_lst_t;
```

4. List Management Structure

The following call back functions are defined in wm_lstTable_t to compare and delete data items from the list
Comparison callback function (CompareItem) is used to compare two items in the list. The comparision function should return a positive value if first element is greater than second and negative when it is less. A zero value should be returned when both elements are equal.

Item destruction callback function (FreeItem) is used to delete items from the list.
The data structure is defined as follows:

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

```
    void    ( * FreeItem ) ( void * );
    } wm_lstTable_t;
```

If the CompareItem and FreeItem callback functions are set to NULL, then the wm_strcmp and wm_osReleaseMemory functions are used by the Open AT Application.

5. List Management APIs

This following section describes the list management APIs available in Open AT OS.

5.1. Creating a List

Use the following API to create a data list in an Open AT Application:

Prototype:

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

Parameters:

Attr: This parameter is used to define the attributes of the created list. A logical OR operation can be performed on the following values to set the value of this parameter:

WM_LIST_NONE: No specific style.
WM_LIST_SORTED: Sorted list.
WM_LIST_NODUPLICATES: Only new elements are added.

funcTable: Pointer to a structure containing the comparison and item destruction callback functions (defined in List Management Structure section).

Returned Values:

Pointer to the created list.

5.2. Destroying a List

Use the following API to clear and destroy a list to release the memory that was used by the list:

Prototype:

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

This API calls the FreeItem callback function to delete each item before destroying the list.

Parameters:

list: The list that must be destroyed.

Returned Values:

None.

5.3. Clearing a List

Use the following API to clear a list:

Prototype:

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

This API calls the FreeItem callback function to delete each item. Unlike the wm_lstDestroy API, the wm_lstClear API does not destroy the list.

Parameters:

list: The list to be cleared.

Returned Values:

None.

5.4. Counting List Items

Use the following API to get the current item count of a particular list:

Prototype:

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

Parameters:

list: The list from which to get the item count.

Returned Values:

Number of items on the specified list.
0 if the list is empty.

5.5. Adding List Items

Use the following API to add an item to the specified list:

Prototype:

s16 wm_lstAddItem (wm_lst_t list, void * item)

Parameters:

list: The list to which you want to add an item.
item: The item that you want to add to the specified list.

Returned Values:

Position of the added item is returned on success.
ERROR is returned if an error occurred while adding the item to the list.

NOTE :
The item pointer should not point to a const because it will be released in an item destruction operation.
If the list has the WM_LIST_SORTED attribute, the item will be inserted at the appropriate place after calling the CompareItem callback (if defined). Otherwise, the item is appended at the end of the list.
If the list has the WM_LIST_ NODUPLICATES attribute, the item will not be inserted if the CompareItem callback function(if defined) returns 0 on any item that was added previously. In this case, the returned index (position) belongs to the existing item.

5.6. Inserting Items into Lists

Use the following API to insert an item to the specified list at the given location:

Prototype:

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

The wm_lstInsertItem API gives the developer the independence to insert a data item at any position in the list, whereas the wm_lstAddItem API appends the data item by itself to the list according to the list attribute defined in the wm_lstCreate API.

Parameters:

list: The list into which you want to insert an item.
item: The item that you want to insert into the specified list.
index: The location on the list at which you want to insert the item.

Returned Values:

Position of the added item is returned on success.
ERROR is returned if an error occurred while inserting the item to the list.

NOTE :
The item pointer should not point to a const as it will be released in an item destruction operation.
This API will always insert the provided item at the given index irrespective of the list attribute.

5.7. Reading List Items

Use the following API to read an item from a given index of the specified list:

Prototype:

```
void * wm_lstGetItem ( wm_lst_t List, s16 index )
```

Parameters:

list: The list from which the item must be read.
index: The location from which the item must be read.

Returned Values:

Pointer to the requested item is returned on success.
NULL is returned if the index is invalid.

5.8. Deleting List Items

Use the following API to delete an item from a given index of the given list:

Prototype:

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

This API calls the FreeItem callback function to delete the specified item.

Parameters:

list: The list from which the item must be deleted.
index: The location from where the item must be deleted.

Returned Values:

Number of items remaining in the list is returned on success.
ERROR is returned if an error occurred.

5.9. Finding a List Item

Use the following API to find an item in the given list:

Prototype:

s16 wm_lstFindItem (wm_lst_t list, void * item)

This API calls the CompareItem callback function on each list item until it returns 0 - indicating that the item has been found.

Parameters:

list: The list you want to search.
item: The item that you want to locate on the specified list.

Returned Values:

Index of the found item is returned on success.
ERROR is returned in all other cases.

5.10. Finding all Items on a List

Use the following API to find all items that match the given data item in the specified list:

Prototype:

s16 * wm_lstFindAllItem (wm_lst_t list, void *item)

This API calls the CompareItem callback function to find all the list items that match a particular item.

Parameters:

list: The list that you want to search.
item: The item that you want to find on the list.

Returned Values:

The s16 buffer holding the indices of all the items that are found. 0 is the first position and ERROR is the last element of this buffer.

NOTE :
The s16 buffer returned by this API must be released by the Open AT Application after it is processed.
This API should be used only if the list must not be changed while processing the returned buffer. Otherwise, the API wm_lstFindNextItem should be used

5.11. Finding the Next Item on a List

Use the following API to find the next item index that corresponds to the provided item in the specified list:

Prototype:

s16 wm_lstFindNextItem (wm_lst_t list, void * item)

This API calls the CompareItem callback function on each list item to get the items that match the provided item. This API should be called until it returns an ERROR in order to get the index of all the items corresponding to the selected item.

Parameters:

list: The list that you want to search.
item: The item that you want to locate on the list.

Returned Values:

Index of the found item is returned on success.
ERROR is returned In all other cases.

NOTE :
The difference between the wm_lstFindNextItem and the wm_lstFindAllItem APIs is that even if the list is updated between two calls to the wm_lstFindNextItem, the function will not return an item that it located on the previous search operation.
To restart a search with the wm_lstFindNextItem, call the wm_lstResetItem first.

5.12. Resetting all Previously Found Items

Use the following API to reset all the items that were previously located by the wm_lstFindNextItem API.

Prototype:

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

This API calls the CompareItem callback on each list item to get the items matching the provided item.
After calling this API, the wm_lstFindNextItem can be used to restart a search for the next item index that corresponds to a given item in the specified list.

Parameters:

list: The list that must be searched.
item: The item that you want to find, in order to reset all the items that were located previously.

Returned Values:

None.

6. Sample Code

```
/* sample code implementing the list management APIs */
#include "adl_global.h"
#if __OAT_API_VERSION__ >= 400
const u16 wm_apmCustomStackSize = 4096;
#else
u32 wm_apmCustomStack[1024];
const u16 wm_apmCustomStackSize = sizeof(wm_apmCustomStack);
#endif
wm_lst_t ListPtr;
u16 Count;
static wm_lstTable_t ListTable =
{
        ( void * ) NULL,
        ( void * ) NULL
};
char Item1[] = "Sierra Wireless";
char Item2[] = "Open-AT";
char Item3[] = "Training";
char Item4[] = "ADL";
void adl_main ( adl_InitType_e InitType )
{
        s16 Position;
        char *buffer;
        u16 index;
TRACE (( 1, "Embedded Application: Main" ));
        ListPtr = wm_lstCreate(WM_LIST_SORTED, &ListTable);
        wm_lstClear (ListPtr);
        Count = wm_lstGetCount(ListPtr);
        TRACE (( 1, "Number of elements in the list = %d", Count ));
        Position = wm_lstAddItem(ListPtr, Item1);
        Position = wm_lstAddItem(ListPtr, Item2);
        Position = wm_lstAddItem(ListPtr, Item3);
        Count = wm_lstGetCount(ListPtr);
        TRACE ((1, "Number of elements in the updated list = %d",Count));
        Position = wm_lstInsertItem(ListPtr, Item4, 1);
        Count++;
        TRACE (( 1, "Item inserted at position %d", Position ));
        for(index=0; index<Count; index++)
        {
                buffer = (char *)wm_lstGetItem(ListPtr, index);
                TRACE (( 1, buffer ));
        }
        Count = wm_lstDeleteItem(ListPtr, --Count);
        Position = wm_lstFindItem(ListPtr, "Open-AT");
```

```
                        TRACE (( 1, "Position of 'Open-AT' = %d", Position ));
                        TRACE (( 1, "Updated list:" ));
                        for(index=0; index<Count; index++)
                        {
                                buffer = (char *)wm_lstGetItem(ListPtr, index);
                                TRACE (( 1, buffer ));
                        }
                        wm_lstDestroy(ListPtr);
        }
```

NOTE :
The wm_lstInsertItem API gives the developer the independence to insert a data item at any position in the list, whereas the wm_lstAddItem API appends the data item by itself to the list according to the list attribute defined in the wm_lstCreate API.
The wm_lstDestroy API clears all the data stored in the list as well as deletes the list. The list pointer cannot be used again after the list is deleted. On the contrary, the wm_lstClear API only clears the contents of list. It does not destroy the list and hence, the list pointer can be used again to add new data to the list.

Summary

The following points have been covered in this chapter

- **Data stored in a list is different from the data stored using flash object IDs. The data stored in the list is lost once an Open AT Application is stopped.**
- **Open AT ADL library directly uses the list management APIs defined for Open AT OS basic APIs. These APIs are declared in the basic API header file wm_list.h.**
- **wm_lstCreate API should be used to create a list.**
- **wm_lstDestroy API should be used to clear and destroy a list.**
- **wm_lstClear API should be used to clear a list.**
- **wm_lstGetCount API should be used to get the current item count of the specified list.**
- **wm_lstAddItem API should be used to add an item to a list.**
- **wm_lstInsertItem API should be used to insert an item to the specified list at the given location.**
- **wm_lstGetItem API should be used to read an item from a list.**
- **wm_lstDeleteItem API should be used to delete an item from a given index of the given list.**
- **wm_lstFindItem API should be used to find an item from a list.**
- **wm_lstFindAllItem API should be used to find all items similar to a given item from a list.**
- **wm_lstFindNextItem API should be used to find the next item in the given list.**
- **wm_lstResetItem API should be used to reset all the previously found items by the wm_lstFindNextItem API**

CHAPTER 28

AT FCM/IO Port Service

1. Objective

This chapter introduces the AT FCM/IO port service and its usage in embedded module. It covers in detail, the APIs and events related to AT FCM/IO service. It also includes prototypes to help you work with this service. This information is provided through the following sections of this chapter:

- Introduction to this service
- AT FCM/IO service related events
- AT FCM/IO service related macros
- AT FCM/IO service related APIs

2. Introduction

AT commands and data can be sent through physical or logical ports.

- Physical ports includes
  - UART1
  - UART2
  - USB
- Logical ports includes
  - CMUX logical ports
  - Bluetooth logical ports
  - GSM/GPRS data ports

To monitor the states of these physical or logical ports, AT FCM/IO port service is used. Using AT FCM/IO service following information can be captured for the physical and logical ports:

- State of port – Opened/Closed
- State of hardware signals DSR and CTS

NOTE :
There is no access to CTS/DSR, when port is opened and flow control is deactivated.

3. AT FCM/IO service related events

The events received in this service are generated from Open AT OS. The following table mentions the events that can be received in the AT FCM/IO service.

Events	**Numeric Value**	**Description**
ADL_PORT_EVENT_OPENED	0	The specified port has opened and now it can be used with AT commands APIs or FCM service.
ADL_PORT_EVENT_CLOSED	1	The specified port has closed now. It should not be used with AT commands APIs or FCM service
ADL_PORT_EVENT_ CTS_STATE_CHANGE	2	The CTS signal for the specified port has been changed
ADL_PORT_EVENT_ DSR_STATE_CHANGE	3	The DSR signal for the specified port has been changed

4. AT FCM/IO service related enums

This section lists the content of enums used by AT FCM/IO service.

4.1. The adl_portSignal_e enum

```
/* Port serial signals */
typedef enum
{
  ADL_PORT_SIGNAL_CTS,    // Serial port CTS input signal
  ADL_PORT_SIGNAL_DSR,    // Serial port DSR input signal
  ADL_PORT_SIGNAL_LAST
} adl_portSignal_e;
```

4.2. The adl_portEvent_e enum

```
/* Port service events */
typedef enum
{
   ADL_PORT_EVENT_OPENED,  // The provided port is now opened
   ADL_PORT_EVENT_CLOSED,  // The provided port is now closed
   // The provided port CTS signal state has changed
   ADL_PORT_EVENT_CTS_STATE_CHANGE,
  // The provided port DSR signal state has changed
   ADL_PORT_EVENT_DSR_STATE_CHANGE
} adl_portEvent_e;
```

5. AT FCM/IO service related Macros

Following macros are available to check whether the port can be used for services such as FCM, AT commands or Bluetooth.

5.1. The ADL_PORT_GET_PHYSICAL_BASE macro

This macro is used to get the physical port of a logical port. For e.g. CMUX logical port is opened over physical port UART1, then this macro will return UART1.

Prototype:

```
adl_port_e ADL_PORT_GET_PHYSICAL_BASE(adl_port_e port)
```

Parameters:

Port: This parameter determines the logical port for which the physical port should be searched. The values this parameter can take are defined in the adl_port_e enum.

Returned values:

Returns the physical port defined by adl_port_e enum.

5.2. The ADL_PORT_IS_A_SIGNAL_CHANGE_EVENT macro

This macro is used to check whether a particular event has occurred due to a transition in the hardware signals such as DSR, CTS or not.

Prototype:

```
bool ADL_PORT_IS_A_SIGNAL_CHANGE_EVENT (adl_portEvent_e Event)
```

Parameters:

Event: This parameter determines the event which should be queried. The values this parameter can take are defined in the adl_portEvent_e enum.

Returned values:

TRUE is returned if the specified event is due to a change in signal.
FALSE is returned if specified event is not due to a change in signal.

5.3. The ADL_PORT_IS_A_PHYSICAL_PORT macro

This macro is used to check whether a particular port is a physical port or not.

Prototype:

```
bool ADL_PORT_IS_A_PHYSICAL_PORT (adl_port_e port)
```

Parameters:

Port: This parameter determines the port which should be queried. The values this parameter can take are defined in the adl_port_e enum.

Returned values:

TRUE is returned if the specified port is physical port.
FALSE is returned if the specified port is virtual port.

5.4. The ADL_PORT_IS_A_PHYSICAL_OR_BT_PORT macro

This macro is used to check whether a particular port is a physical port or a Bluetooth port.

Prototype:

```
bool ADL_PORT_IS_A_PHYSICAL_OR_BT_PORT (adl_port_e port)
```

Parameters:

Port: This parameter determines the port which should be queried. The values this parameter can take are defined in the adl_port_e enum.

Returned values:

TRUE is returned if the specified port is physical or Bluetooth port.
FALSE is returned if the specified port is not a physical or Bluetooth port.

5.5. The ADL_PORT_IS_AN_FCM_PORT macro

This macro is used to check whether a particular port can be used by Flow control manager or not.

Prototype:

```
bool ADL_PORT_IS_AN_FCM_PORT (adl_port_e port)
```

Parameters:

Port: This parameter determines the port which should be queried. The values this parameter can take are defined in the adl_port_e enum.

Returned values:

TRUE is returned if the specified port can be used by FCM service.
FALSE is returned f the specified port cannot be used by FCM service.

5.6. The ADL_PORT_IS_AN_AT_PORT macro

This macro is used to check whether a particular port can be used by AT commands APIs or not.

Prototype:

```
bool ADL_PORT_IS_AN_AT_PORT (adl_port_e port)
```

Parameters:

Port: This parameter determines the port which should be queried. The values this parameter can take are defined in the adl_port_e enum

Returned values:

TRUE is returned if the specified port can be used by AT commands APIs.
FALSE is returned if the specified port cannot be used by AT commands APIs.

6. AT FCM/IO service related APIs

Following is the list of APIs provided by this service:

6.1. The adl_portSubscribe API

This API is used to subscribe to port service. The subscription allows Open AT Application to monitor the events related to AT FCM/IO service.

Prototype:

```
s8 adl_portSubscribe ( adl_portHdlr_f PortHandler )
```

Parameters:

PortHandler: This is the call back handler, which receives the events related to AT FCM/IO service. This handler should be defined as shown below:

typedef void (* adl_portHdlr_f) (adl_portEvent_e Event, adl_port_e Port, u8 State)

Parameters of the call back handler:

- **Event:** Event (Port Opened/Closed) that has occurred for a particular port. The values this parameter can take are defined in the adl_portEvent_e enum
- **Port:** Port for which the event has occurred. The values this parameter can take are defined in the adl_port_e enum
- **State:** This is set when there is a change in the signal state. This parameter gives the state of changed signal.
-

Returned values:

Positive or null Handle is returned on success.
ADL_RET_ERR_PARAM is returned if there is an error in the parameter.
ADL_RET_ERR_NO_MORE_HANDLES is returned if there are no more free handlers.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

NOTE :
Only 127 subscriptions are allowed in the AT FCM/IO service.

6.2. The adl_portUnsubscribe API

This API is used to unsubscribe from port service.

Prototype:

s8 adl_portUnsubscribe (u8 Handle)

Parameters:

Handle: Handle returned by adl_portSubscribe() API

Returned values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if unknown handle is passed.
ADL_RET_ERR_NOT_SUBSCRIBED is returned if the service is not subscribed.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

6.3. The adl_portIsAvailable API

This API is used to check if the particular port is opened or not.

Prototype:

bool adl_portIsAvailable (adl_port_e Port)

Parameters:

Port: This parameter determines the port which should be queried. The values this parameter can take are defined in the adl_port_e enum.

Returned values:

TRUE is returned if port is opened.
FALSE is returned if port is not opened.

6.4. The adl_portGetSignalState API

This API is used to get the signal state of a particular signal (CTS, DSR).

Prototype:

```
s8 adl_portGetSignalState ( adl_port_e Port, adl_portSignal_e Signal )
```

Parameters:

Port: This parameter determines the port which should be queried. The values this parameter can take are defined in the adl_port_e enum.
Signal: The signal which should be queried for the state. The values this parameter can take are defined in the adl_portSignal_e enum.

Returned values:

State of the signal (0/1) is returned if a valid signal is queried.
ADL_RET_ERR_PARAM is returned if there is an error in the parameter.
ADL_RET_ERR_BAD_STATE is returned if the port is not opened.

6.5. The adl_portStartSignalPolling API

This API is used to poll for a particular signal (CTS/DSR) of a port.

Prototype:

```
s8 adl_portStartSignalPolling ( u8 Handle, adl_port_e Port, adl_portSignal_e Signal, u8 PollingTimerType, u32 PollingTimerValue )
```

Parameters:

Handle: Handle retuned by adl_portSubscribe() API.
Port: This parameter determines the port which should be queried. The values this parameter can take are defined in the adl_port_e enum.

Signal: The signal which should be polled for the state. The values this parameter can take are defined in the adl_portSignal_e enum. This enum is defined in the section 22.4.1.
PollingTimerType: This parameter determined the type of the time that should be used for polling. The type could be one of the following:
ADL_TMR_TYPE_100MS – 100 ms timer
ADL_TMR_TYPE_TICK – 18.5 ms time
PollingTimerValue: The polling timer value in terms of PollingTimerType

Returned values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if there is an error in the parameter.
ADL_RET_ERR_BAD_STATE is returned if the port is not opened.
ADL_RET_ERR_UNKNOWN_HDL is returned if unknown handle is passed.
ADL_RET_ERR_NOT_SUBSCRIBED is returned if the service is not subscribed.
ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if a polling process has already started.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

6.6. The adl_portStopSignalPolling API

This API is used to stop the polling of a particular signal of a port.

Prototype:

```
s8 adl_portStopSignalPolling ( u8 Handle, adl_port_e Port, adl_portSignal_e Signal )
```

Parameters:

Handle: Handle retuned by adl_portSubscribe () API.
Port: This parameter determines the port which was used during subscription. The values this parameter can take are defined in the adl_port_e enum.
Signal: The signal which should was used during subscription. The values this parameter can take are defined in the adl_portSignal_e enum.

Returned values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if there is an error in the parameter.
ADL_RET_ERR_BAD_STATE is returned if the port is not opened.
ADL_RET_ERR_UNKNOWN_HDL is returned if unknown handle is passed.
ADL_RET_ERR_BAD_HDL is returned if no polling process is running.
ADL_RET_ERR_NOT_SUBSCRIBED is returned if the service is not subscribed.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

7. Sample Code

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 4096;
```

```
/* Callback function for Port service */
void PortHandler ( adl_portEvent_e Event, adl_port_e Port, u8 State )
{

  TRACE (( 1, "PortHandler: Event %d, Port %d, State %d", Event, Port, State ));

  if( Event == ADL_PORT_EVENT_CTS_STATE_CHANGE )
    adl_atSendResponsePort(ADL_AT_RSP, ADL_PORT_UART1, "CTS Changed");
}

void adl_main ( adl_InitType_e InitType )
{
  s8 sRet,sPortHandle;
  TRACE (( 1, "Embedded Application: Main" ));
  /* Subscribe to port service to poll for CTS signal */
  sPortHandle = adl_portSubscribe ( PortHandler );
  TRACE (( 1, "adl_portSubscribe: %d", sPortHandle ));
  /* Start the polling service to monitor the state of CTS signal */
  sRet = adl_portStartSignalPolling ( sPortHandle, ADL_PORT_UART1,
                                     ADL_PORT_SIGNAL_CTS,
                                     ADL_TMR_TYPE_TICK, 1);
}
```

Summary

The following points have been covered in this section

- **AT FCM/IO service can be used to check whether a flow can be used by AT or FCM service or not.**
- **This service allows to poll for hardware signals such as CTR, DSR.**

CHAPTER 29

RTC Service

1. Objective

This chapter introduces the RTC service and its usage in embedded module. It covers in detail the APIs and events related to RTC, and also include prototypes to help you work with this service. This information is provided through the following sections of this chapter:

- **Introduction to this service**
- **RTC service related APIs**

2. Introduction

The RTC service allows Open AT Application to access the internal clock provided by embedded module. This service also allows converting the time to following formats:

- Time in (YY/MM/DD/HH/MM/SS) format
- Time in UNIX format (Seconds elapsed since 1st January 1970)

3. Macros and Constants

Macro/Constant	Description
ADL_RTC_DAY_SECONDS	Seconds count in a day
ADL_RTC_HOUR_SECONDS	Seconds count in a hour
ADL_RTC_MINUTE_SECONDS	Seconds count in a minute
ADL_RTC_MS_US	µseconds count in a millisecond
ADL_RTC_SECOND_FRACPART_STEP	Second fractional part step value (in µs)
ADL_RTC_GET_TIMESTAMP_DAYS	From a TimeStamp date extract the days number
ADL_RTC_GET_TIMESTAMP_HOURS	From a TimeStamp date extract the hours number
ADL_RTC_GET_TIMESTAMP_MINUTES	From a TimeStamp date extract the minutes number
ADL_RTC_GET_TIMESTAMP_SECONDS	From a TimeStamp date extract the seconds number

Macro/Constant	Description
ADL_RTC_GET_TIMESTAMP_MS	From a TimeStamp date extract the milliseconds number
ADL_RTC_GET_TIMESTAMP_US	From a TimeStamp date extract the µseconds number

4. RTC service related APIs

Following is the list of APIs provided by this service:

4.1. Retreiving Second Fractional Part

This API is used to retrieves the second fractional part step (in µs), reading the rtc_PreScalerMaxValue register field.

Prototype:

```
float adl_rtcGetSecondFracPartStep ( void )
```

Parameters:

None

Returned values:

The second fractional part step of the embedded module, in µs.

4.2. Retreving the Current RTC Time

This API is used to get the current time of the internal RTC in (YY/MM/DD/HH/MM/SS) format.

Prototype:

```
s32 adl_rtcGetTime ( adl_rtcTime_t * TimeStructure )
```

Parameters:

TimeStructure: The structure which will be filled by ADL with RTC data. This structure is defined below:

```
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;
```

Returned values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if there is an error in the parameter.

4.3. Configuring the RTC Time

This API is used to set the current time of the internal RTC in (YY/MM/DD/HH/MM/SS) format.

Prototype:

```
s32 adl_rtcSetTime ( adl_rtcTime_t * TimeStructure )
```

Parameters:

TimeStructure: The structure which has the RTC data to set. This structure is defined below:

```
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;
```

Returned values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if there is an error in the parameter.

4.4. Converting the RTC Time

This API is used to convert the time to the UNIX or YY/MM/DD/HH/MM/SS format.

Prototype:

```
s32 adl_rtcConvertTime ( adl_rtcTime_t * TimeStructure, adl_rtcTimeStamp_t * TimeStamp, adl_rtcConvert_e Conversion )
```

Parameters:

TimeStructure: Time in YY/MM/DD/HH/MM/SS format. This structure which will be converted or filled with converted RTC data depending upon the Conversion parameter. This structure is defined below:

```
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;
```

TimeStamp: Time in UNIX format. This structure which will be converted or filled with converted RTC data depending upon the Conversion parameter. This structure is defined below:

```
typedef struct
{
   u32 TimeStamp;      // Seconds elapsed since 1st January 1970
   u32 SecondFracPart; // Second fractional part (0-32767)
} adl_rtcTimeStamp_t;
```

Conversion: This parameter determines whether time will be converted to UNIX or YY/MM/DD/HH/MM/SS format. The value this parameter can take are defined below in the adl_rtcConvert_e enum:

```
// Conversion modes
typedef enum
{
   ADL_RTC_CONVERT_TO_TIMESTAMP,   // RTC Time Structure to
                                   //  TimeStamp conversion
   ADL_RTC_CONVERT_FROM_TIMESTAMP, // TimeStamp to RTC Time
                                   //  Structure conversion
} adl_rtcConvert_e;
```

Returned values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if there is an error in the parameter.
ERROR is returned in case of an internal error

4.5. Finding the Difference between two UNIX based TimeStamps

This API is used to find the difference between two UNIX based timestamps.

Prototype:

```
s32 adl_rtcDiffTime ( adl_rtcTimeStamp_t * TimeStamp1, adl_rtcTimeStamp_t * TimeStamp2, adl_rtcTimeStamp_t * Result )
```

Parameters:

TimeStamp1: This is the first UNIX based Timestamp used for comparision. The value this parameter can take are defined in the adl_rtcTimeStamp_t structure. Please refer section 28.3.4 for description of this structure.
TimeStamp2: This is the second UNIX based Timestamp used for comparison. The values this parameter can take are defined in the adl_rtcTimeStamp_t structure. Please refer section **Error! Reference source not found.** for description of this structure.
Result: Result of the difference between TimeStamp1 and TimeStamp2. The value this parameter can take are defined in the adl_rtcTimeStamp_t structure. Please refer section 28.3.4 for description of this structure.

Returned values:

1 is returned if TimeStamp1 is greater than TimeStamp2
-1 is returned if TimeStamp2 is greater than TimeStamp1
0 is returned if TimeStamp1 is equal to TimeStamp2
ADL_RET_ERR_PARAM is returned if there is an error in the parameter.

5. Sample Code

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 1024

void adl_main ( adl_InitType_e InitType )
{
   /* Local Variables */
               adl_rtcTime_t                          tTime;                    /* To store the current time */
   adl_rtcTimeStamp_t              tTimeStamp; /* To store the time in Unix format */
               ascii RspBuffer[100];        /* To prepare response for serial port */
               s32 sRet;
               TRACE (( 1, "Embedded Application: Main" ));

               /* Get the RTC time */
               sRet = adl_rtcGetTime( &tTime );

               /* In case of an error, send the same on serial port */
               if ( sRet != OK )
               {
                              adl_atSendResponsePort(ADL_AT_UNS,ADL_PORT_UART1,"\r\nError fetching RTC
                                                            time\r\n" );
                              return;
               }

               /* Prepare the buffer for serial port */
               wm_memset( RspBuffer, 0x00, (sizeof(RspBuffer) / (sizeof (ascii))) );
               wm_sprintf( RspBuffer, "\r\nRTC Time (YYYY/MM/DD - HH:MM:SS):" \
                              " %.4d/%.2d/%.2d - %.2d:%.2d:%.2d\r\n", tTime.Year, tTime.Month,
                              tTime.Day, tTime.Hour, tTime.Minute, tTime.Second );
               /* Print the current RTC time to serial port */
               adl_atSendResponsePort( ADL_AT_UNS, ADL_PORT_UART1 , RspBuffer );

               /* Convert the time in Unix format */
               sRet = adl_rtcConvertTime ( &tTime, &tTimeStamp, ADL_RTC_CONVERT_TO_TIMESTAMP );

               /* In case of an error, send the same on serial port */
               if ( sRet != OK )
               {
                              adl_atSendResponsePort ( ADL_AT_UNS, ADL_PORT_UART1,"\r\nError converting
                                                            RTC time\r\n");
                              return;
               }

               /* Prepare the buffer for serial port */
               wm_memset( RspBuffer, 0x00, (sizeof(RspBuffer) / (sizeof (ascii))) );
               wm_sprintf( RspBuffer, "\r\nRTC Unix Time - Seconds elapsed since" \
                              " 1st January 1970: %d\r\n", tTimeStamp.TimeStamp );

                /* Print the current RTC Unix time to serial port */
                adl_atSendResponsePort( ADL_AT_UNS, ADL_AT_PORT1, RspBuffer );
}
```

Summary

The following points have been covered in this section

- **RTC APIs can be used to access internal RTC time**
- **RTC time is available in two formats**
  - **UNIX format**
  - **YY/MM/DD/HH/MM/SS format**

CHAPTER 30

Real Time Operating System

1. Introduction to RTOS

RTOS is an operating system, which provides a system, which guarantees a certain capability within a specified time constraint. RTOS provides facilities which, if used properly, guarantee deadlines can be met generally (soft real-time) or deterministically (hard real-time). . This amount of accuracy is achieved through specialized scheduling algorithms. The main criterion for an RTOS is to schedule tasks in minimal latency.

A real-time system is judged based on following three parameters:

- Determinism: This means system takes same amount of time to a certain work. For e.g. A mobile phone could be called a real-time system, if it can place a call whenever a call button is pressed and each time it takes fixed amount of time to place the call let's say 10 milliseconds.

- Timeliness: This means system completes the time meeting the deadlines. For e.g. in the above example, 10 milliseconds – 1 second could be an acceptable time for the user. However if the mobile phone takes more than 1 second to place the call, it cannot be called a real-time system.

- Predictability: This means the output from the system is predictable. This parameter is again related to determinism and timeliness. For e.g. you can predict the amount of time mobile phone could take to place the call.
- A Real-time operating system will provide specialized features to cater to the needs of above mentioned parameters. The features are listed below:

- Multitasking: A system has needs for different applications. For e.g. telemetry system has one application logic to communicate with the server and application logic for data acquisition. RTOS provides multitasking so that these different applications can run in parallel. These tasks can have different configurable priorities which is very useful to decide the real-time nature of the system.

- Scheduling Algorithms: RTOS provides specialised scheduling algorithms to allow the system to be deterministic. A RTOS must have facility to configure these scheduling algorithms to help designing a real-time system. These algorithms could be priority based or time based. RTOS should take fixed amount of time to execute these algorithms and this time should be known to the user. This allows systems to be more predictable.
- This is the main difference between a general purpose OS and RTOS. In a operating system like Windows, it is difficult to predict the time. For e.g. A notepad application could take different

amount of time to save a file of the same size. However in a RTOS, the time will be always the same and hence providing much more predictable system.

- Inter-process communication: It is highly likely that tasks will interact with each other in a multi-tasking system. This interaction could be one of the following:
- Communication: To transfer information amongst tasks.
- Synchronisation: To co-ordinate a sequence of events amongst tasks.
- Mutual exclusion: To access a shared resource
- Interrupts and high-precision timers: RTOS allows developer to control the communication with various devices through interrupts and also allows controlling the flow of application through high precision timers.

You can create intelligent internet-enabled applications directly on the embedded module. This shortens development time and reduce materials cost. However earlier, it was not possible for developer to write time critical application along with GSM/GPRS capabilities. To overcome the limitation, Sierra Wireless has introduced the feature "RTOS".
RTOS feature guarantees task execution within a specified time constraint. This feature allows user to:

- Access to various interrupts such as Bus, Audio, Timer, Pins.
- Use of high granularity timers (<1ms).
- Access to low level interruption handlers.
- Access to high level interruption l   handlers.
- Multiple tasks and synchronization using semaphores.
- Configuration of processor clock speed.
- Predictable response time at each priority level.

This feature is available from Firmware 6.61 and Open AT OS v4.10.

NOTE :
AT+WCFM command should be used to enable RTOS feature.

2. Open AT RTOS Architecture

The task management done by Open AT OS has been changed for incorporating RTOS feature. The following figure depicts various tasks that are managed in embedded module.

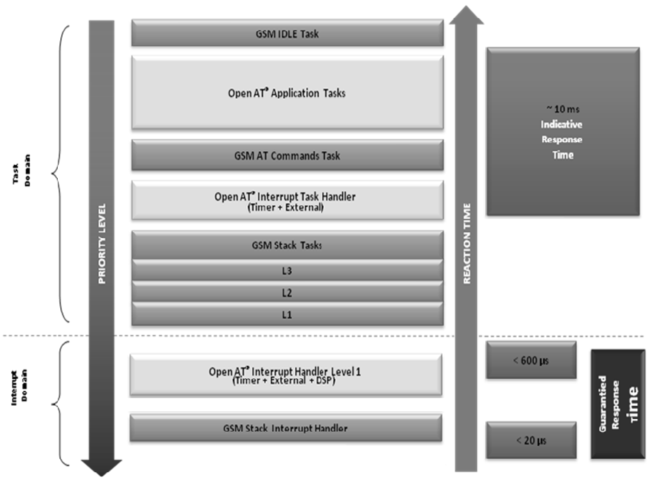

Task Domain
Interrupt Domain
PRIORITY LEVEL
GSM IDLE Task
Open AT® Application Tasks
GSM AT Commands Task
Open AT® Interrupt Task Handler (Timer + External)
GSM Stack Tasks
L3
L2
L1
Open AT® Interrupt Handler Level 1 (Timer + External + DSP)
GSM Stack Interrupt Handler
REACTION TIME
~10 ms Indicative Response Time
< 600 µs
< 20 µs
Guarantied Response Time

Figure 73 - Architecture of RTOS

Using Open AT OS, you can write your code in following areas:

- Low level Interrupt handlers
- High level interrupt handlers
- Open AT Applications

The Open AT Interrupt handler Level 1 has the highest priority than Firmware tasks. This Interrupt handler can be attached to one of the following interrupts

- External interrupts
- Internal timer interrupts
- DSP (Audio) interrupts

Following table lists the response time of various handlers and applications.

Handler/Application	Response Time	Description
Open AT Interrupt handler Level 1	Less than 600 µs	Please make sure that low-level handlers have minimum possible code so that it does not have adverse effects on other low priority tasks
Open AT Interrupt Task handler	1 to 10 ms	The reaction time of this hander could go up to 1 ms to 10ms depending upon coding of the application. These handlers should be used when a part of code needs to be executed before Open AT Applications
Open AT Application Task (s)	Approximately 10 ms	The reaction time depends on the processing done by the Firmware tasks and interrupt handlers. You can write up to 64 tasks in this area.

3. Scheduling policies

The Open AT OS implements following scheduling policy:

Fully pre-emptive
In fully pre-emptive policy, the higher priority task/interrupt handler will preempt the low priority task execution. The context of the low priority will be saved during this time and same will be restored when the task resumes again.

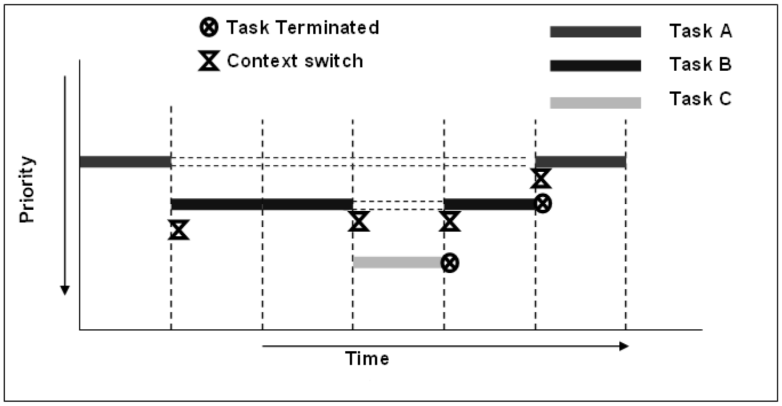


Figure 74 - Task Pre-emption

4. RTOS APIs

To manage different interrupt handlers and applications, Open AT OS RTOS feature provides following set of APIs:

- Task Management
  - Multitasking
  - Context
- Inter-task communication
  - Message service
- Synchronization
  - Semaphore service
- Thread safe data management
  - Queue service
- Interrupts
  - IRQ
  - External interrupts
  - Timer interrupts – TCU service
  - Audio interrupts
- Vari-speed

NOTE :
Currently service for UART interrupts and ADC interrupts are not available.

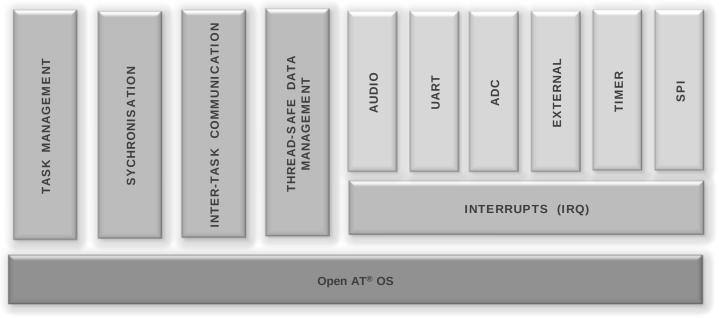


Figure 75 - RTOS API Architecture

These APIs are discussed in detail in chapters from 0 to **Error! Reference source not found.**.

Summary

The following points have been covered in this chapter

- **RTOS feature guarantees task execution within a specified time constraint.**
- **Using Open AT OS, user can write applications in following areas:**
  - **Task Management**
    - **Multitasking**
    - **Context**
  - **Inter-task communication**
    - **Message service**
  - **Synchronization**
    - **Semaphore service**
  - **Thread safe data management**
    - **Queue service**
  - **Interrupts**
    - **IRQ service**
    - **External interrupts**
    - **Timer interrupts – TCU service**
    - **Audio interrupts**
  - **VariSpeed**

5. Task Management – Multitasking API's

This chapter introduces the multitasking APIs provided by Open AT OS.

5.1. Introduction

Multitasking is a process by which multiple tasks share common processing resources and allows multiple tasks to run at the same time.

User can define 64 tasks in an Open AT Application and the communication between tasks can be done using:

- Queues
- Shared Memory Area
- Mailboxes

Task size is defined by the user within the RAM area available for Open AT Application. Typical size is up to 4KBytes per task.

5.2. Structure to Define Tasks

Open AT OS allows defining multiple tasks. Each task is defined 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

Each task is defined using the following structure in Open AT Application.

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

Parameters:

EntryPoint: This is the entry point for the task. This is called each time when the embedded module reboots and as soon as the Open AT Application is in execution.

StackSize: This parameter is used to provide 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 Applicaion.

Name: This is a string parameter which identifies the task. This is usually used for debugging purposes.

Priority: This parameter defines the priority of the task. The priority level defined in this table should be from 1 to the task count. This priority determines the order in which the events are notified to the several tasks when several information are received at the same time. The higher the priority level, higher the priority of the task.

NOTE :
Last entry in the adl_InitTasks_t table must be set to NULL. Two tasks can't have the same priority level. Maximum of 64 tasks can be defined.

5.3. Sample Code

```
const  adl_InitTasks_t adl_InitTasks [] =
{
 {SmsTask, 1024, “SMS control Task”, 1},
 {CallTask, 1024, “Call control Task”, 2},
 {MainTask, 8192, “Master Task”,3},
 {0,0,0,0}
}
```

6. Task Management - Context Service APIs

6.1. Introduction

In multi-tasking environment, when a piece of code is being executed from several tasks/interrupts, context APIs can be used to get the information about the current task context. This information could be useful to re-use a piece of code for all the possible tasks/interrupts for e.g. a library providing a set of APIs could use context APIs.
Open AT OS provides the context service interface that allows to get the task information at run time. This information includes:

- The current execution context
- Number of tasks running
- Current task state

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. This context has a global low priority and can be interrupted by any other tasks.
- The high level interrupt handler context: This is also a task context, but with a higher priority than the main application task. High level interrupt handlers run in this context. This context has a global middle priority and can be interrupted by low level interrupt handler context. 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.
- 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 and can't be interrupted by any other tasks. When an interrupt occurs, this context will be immediately activated, even if an application task or high level handler or an Firmware task is running. Note that the time spent in this context should be as short as possible.

The header file adl_ctx.h should be included to use the context service APIs.

6.2. Get the Current Context

This API can be used to get the current context of the embedded module. The context could be from:

- Open AT Application
- Low level interruption handler
- High level interruption handler

Prototype:

```
adl_ctxID_e adl_ctxGetID ( void )
```

Parameters:

None

Returned Values:

Context identifier of type adl_ctxID_e is returned on success. The adl_ctxID_e enum is defined below:

```
typedef enum _adl_ctxID_e
{
  // Low Level Interruption Handler context
  ADL_CTX_LOW_LEVEL_IRQ_HANDLER =0xFD,
  // High Level Interruption Handler context
  ADL_CTX_HIGH_LEVEL_IRQ_HANDLER=0xFE,
  // Reserved for internal use
  ADL_CTX_ALL = 0xFF,
  // Firmware tasks context
  ADL_CTX_WAVECOM = 0xFF,
} adl_ctxID_e;
```

ID of an application task's zero-based index if the function is called from an ADL service event handler.
ADL_CTX_LOW_LEVEL_IRQ_HANDLER is returned if the function is called from a low level interrupt handler.
ADL_CTX_HIGH_LEVEL_IRQ_HANDLER is returned if the function is called from a high level interrupt handler.

6.3. Get the Current Task

This API can be used to get current task identifier.
This function behaves like the adl_ctxGetID API when if it is called from Open AT task or high level interrupt handler context.
When this function is called from the 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 )
```

Parameters:

None

Returned Values:

Context identifier of type adl_ctxID_e is returned on success. The adl_ctxID_e enum is defined below:

```
typedef enum _adl_ctxID_e
{
  // Low Level Interruption Handler context
  ADL_CTX_LOW_LEVEL_IRQ_HANDLER =0xFD,
  // High Level Interruption Handler context
  ADL_CTX_HIGH_LEVEL_IRQ_HANDLER=0xFE,
  // Reserved for internal use
  ADL_CTX_ALL = 0xFF,
  // Firmware tasks context
  ADL_CTX_WAVECOM = 0xFF,
} adl_ctxID_e;
```

ID of an application task's zero-based index if the function is called from an ADL service event handler.
ADL_CTX_HIGH_LEVEL_IRQ_HANDLER is returned if the function is called from a high level interrupt handler.
Interrupted task ID is returned if the function is called from a low level interrupt handler. The returned value depends on the interrupted task:
An application task's zero-based index is returned if an Open AT Application task was running.
ADL_CTX_WAVECOM is returned if a Firmware task was running.
ADL_CTX_HIGH_LEVEL_IRQ_HANDLER is returned if a high level interrupt handler was running.

6.4. Get the Current Tasks Count

This API can be used to get the count of current tasks in the application.

Prototype:

```
u8 adl_ctxGetTasksCount ( void )
```

Parameters:

None

Returned Values:

This function returns current application task count.

6.5. Get the Diagnostic Information

The API can be used to get the information about the current application's execution contexts.

Prototype:

```
u32 adl_ctxGetDiagnostic ( void )
```

Parameters:

None

Returned Values:

Bitwise OR combination of the diagnostics listed in the adl_ctxDiagnostic_e type is returned on success. The adl_ctxDiagnostic_e is defined below:

```
  typedef enum
```

```
{
  // The Open AT IRQ processing mechanism has not been started (interrupt
  // handlers stack sizes have not been supplied).
  ADL_CTX_DIAG_NO_IRQ_PROCESSING= 0x01,
  // Reserved for future use
ADL_CTX_DIAG_BAD_IRQ_PARAM= 0x02,
// High level interrupt handler is not supported
  ADL_CTX_DIAG_NO_HIGH_LEVEL_IRQ_HANDLER= 0x04,
} adl_ctxDiagnostic_e;
```

6.6. Get the State of the Context

This API can be used to get the current state of the context.

Prototype:

```
s32 adl_ctxGetState (adl_ctxID_e Context )
```

Parameters:

Context: Execution context from which the current state has to be queried

Returned Values:

Current execution context state (positive or null) of adl_ctxState type is returned on success.

```
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;
```

ADL_RET_ERR_PARAM is returned on parameter error.
ADL_RET_ERR_BAD_HDL is returned 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):
1 the low level interrupt handler execution context (in any case)
2. the high level interrupt handler execution context, if the related adl_InitIRQHighLevelStackSize call stack has not be declared in the application.

6.7. Suspend a Task

This API can be used to suspend a task execution. This method could be useful to perform synchronization amongst multiple tasks. For e.g. When a task is waiting for a resource owned by another task, it could suspend itself. When the other task release the required resource, it can resume the first task.

Prototype:

s32 adl_ctxSuspend (adl_ctxID_e Task)

Parameters:

Task: Task identifier to be suspended.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned on parameter error.
ADL_RET_ERR_BAD_STATE is returned if the required task is already suspended.

NOTE :
The events which are received when the task is suspended are queued until the task is resumed. If too many events occur, the application mailbox would be overloaded, and this would lead the embedded module to reset.

6.8. Suspend Multiple Tasks

This API can be used to suspend several task's execution.

Prototype:

s32 adl_ctxSuspendExt (u32 TasksCount, adl_ctxID_e * TasksIDArray)

Parameters:

TasksCount: Number of tasks to be suspended.
TaskIDArray: Array containing the identifiers of the tasks to be suspended.

Returned Values:

OK is returned on success.

ADL_RET_ERR_BAD_STATE is returned if the required task is already suspended.
ADL_RET_ERR_PARAM is returned on parameter error.

NOTE : The events which are received when the task is suspended are queued until the task is resumed. If too many events occur, the application mailbox would be overloaded, and this would lead the embedded module to reset.

6.9. Resume a Task

This API is used to resume a task suspended using adl_CtxSuspend() API.

Prototype:

```
s32 adl_ctxResume ( adl_ctxID_e Task )
```

Parameters:

Task: Tasks identifier to be suspended**.**

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned on parameter error.
ADL_RET_ERR_BAD_STATE is returned if the required task is not currently suspended.

NOTE :
1. The required task is resumed as soon as the function is called.
2. 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.
3. If the resumed task has a higher priority level than the current one, it will be scheduled as soon as the function is called.

6.10. Resume Multiple Tasks

This API is used to resume multiple tasks suspended using adl_CtxSuspendExt API.

Prototype:

```
s32 adl_ctxResumeExt (u32 TasksCount , adl_ctxID_e * TasksIDArray )
```

Parameters:

TasksCount: Number of tasks to be resumed.
TaskIDArray: Array of tasks identifiers to be resumed.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned on parameter error.
ADL_RET_ERR_BAD_STATE is returned if the required task is not currently suspended.

NOTE :
1. The required task is resumed as soon as the function is called.
2. 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.
3. If the resumed task has a higher priority level than the current one, it will be scheduled as soon as the function is called.

6.11. Sleep a Task

This API is used to put the current execution context to sleep for the required duration. The 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 (18.5 ms).

Returned Values:

OK is returned on success (when the function returns, the sleep duration has already elapsed).
ADL_RET_ERR_SERVICE_LOCKED is returned f the function was called from a low level interrupt handler (the function is forbidden in this context).

6.12. Sample Code

```
// 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 );
          }
```

Summary

The following points have been covered in this chapter

- **Multitasking is a process by which multiple tasks share common processing resources and allows multiple tasks to run at the same time.**
- **User can define 64 tasks in Open AT Application and the communication between tasks can be done using:**
  - **Queues**
  - **Shared Memory Area**
  - **Mailboxes**
- **Task size is defined by the user within the RAM area available for Open AT Application. Typical size is up to 4KBytes per tasks**
- **In multi-tasking environment, when a piece of code is being executed from several tasks/interrupts, context APIs can be used to get the information about the current task context.**
- **Following APIs are available for context service:**
  - **adl_ctxGetID to get the current context id**
  - **adl_ctxGetTasksCount to get the current tasks count**
  - **adl_ctxGetTaskID to get the current task id**
  - **adl_ctxGetDiagnostic to get the diagnostic information**
  - **adl_ctxSuspend to suspend the task**
  - **adl_ctxSuspendExt to suspend several tasks**
  - **adl_ctxResume to resume a task**
  - **adl_ctxResumeExt to resume several tasks**
  - **adl_ctxSleep to put a task to sleep for a specified duration**

7. Inter-task Communication – Message Service APIs

This chapter introduces the inter-task communication APIs provided by Open AT OS.

7.1. Introduction

Open AT OS provides message service which can be used for communication amongst tasks. Each task is provided with its own mailbox and the messages are exchanged using mailbox identifier as shown in the below diagram.

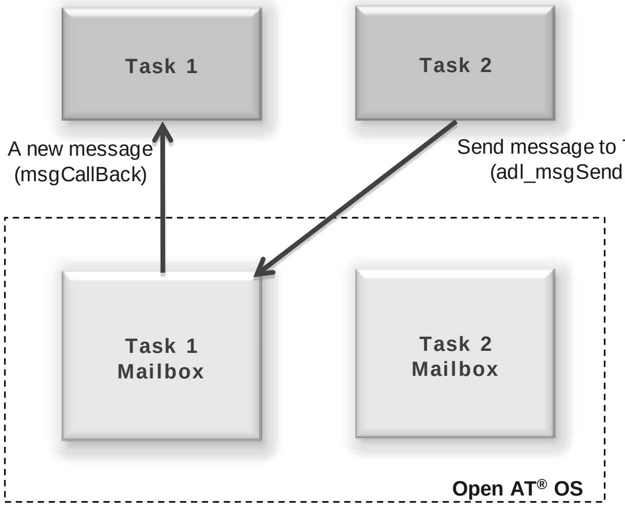

Task 1
Task 2
A new message (msgCallBack)
Send message to Task1 (adl_msgSend())
Task 1 Mailbox
Task 2 Mailbox
Open AT® OS

Figure 76 - Inter-Task Communication Example

7.2. Header file

ADL provides the Message service interface that allows to exchange message between tasks. The header file adl_msg.h should be included in the Open AT Application to use this service.

7.3. Subscribe to Message Service

This API can be used to subscribe to the message service. This function allows the Open AT Application to receive incoming user-defined messages, sent from the application task itself or interrupt handlers.

Prototype:

```
s32 adl_msgSubscribe (adl_msgFilter_t_ * Filter, adl_msgHandler_f msgHandler)
```

Parameters:

Filter: This parameter defines the identifier and source context conditions to check each message reception in order to notify the message handler. This is defined by the structure as mentioned below:

```
typedef struct
{
 u32 MsgIdentifierMask;
 u32 MsgIdentifierValue;
```

```
  adl__msgIdComparator_e Comparator;
  adl__ctxID_e Source;
} adl_msgFilter_t;
```

Parameters:

MsgIdentifierMask: This parameter defines the 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: This parameter defines the message identifier value to be compared with the received message identifier. Only the bits filtered by the MsgIdentifierMask mask are significant.

Comparator: This parameter defines the operator to be used for comparing the received message identifier. This is defined by the adl_msgIdComparator_e as mentioned below:

```
typedef enum
{
  //Two identifiers are equal
  ADL_MSG_ID_COMP_EQUAL,
  //Two identifiers are different
  ADL_MSG_ID_COMP_DIFFERENT,
  //Received message identifier is greater than the subscribed
  //message identifier
  ADL_MSG_ID_COMP_GREATER,
  //Received message identifier is greater or equal to the subscribed
  //message identifier
  ADL_MSG_ID_COMP_GREATER_OR_EQUAL,
  //Received message identifier is lower than the subscribed message
  //identifier.
  ADL_MSG_ID_COMP_LOWER,
  //Received message identifier is lower or equal to the subscribed message
  //identifier.
  ADL_MSG_ID_COMP_LOWER_OR_EQUAL,
  //Reserved for internal use
  ADL_MSG_ID_COMP_LAST,
} adl_msgIdComparator_e;
```

Source: This parameter defines the incoming message source context. The ADL_CTX_ALL constant should be used if the application wishes to receive all messages.

msgHandler: This parameter defines the message handler which will be notified each time a message is received which matches with the filter conditions.

Prototype:

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

Parameters:
MsgIdentifier: This is an identifier for the incoming message.
Source: This is the context of the source from which message has been received.
Length: This is the length of the message body. This is set to 0 in case the message doesn't have the body.
Data: This is the address of the message body. This is set to NULL in case the message doesn't have the body.

Returned Values:

Positive or null value will be returned on success.
Negative value is returned in case of an error.
ADL_RET_ERR_PARAM will be returned if a parameter has an incorrect value.
ADL_RET_ERR_SERVICE_LOCKED will be retuned if the function is called from a low level Interrupt handler.

7.4. Unsubscribe from Message Service

This API can be used to unsubscribe from the message service. The message handler will not receive any message if the un-subscription is successful.

Prototype:

```
s32 adl_msgUnsubscribe (s32 MsgHandle )
```

Parameters:

MsgHandle: This is the handle returned by the adl_msgSubscribe API.

Returned Values:

OK will be returned on success.
ADL_RET_ERR_UNKNOWN_HDL will be returned if the supplied handle is unknown.
ADL_RET_ERR_SERVICE_LOCKED will be returned if the function was called from a low level Interrupt handler.

7.5. Sending Message to Task

This API can be used to send message at any time to any running task. When a message is posted, the source context identifier is automatically set accordingly to the current context:
If the message is sent from the application task, the source context identifier is set to task identifier of the sending task.
If the message is sent from a low level Interrupt handler, the source context identifier is set to ADL_CTX_LOW_LEVEL_IRQ_HANDLER.
If the message is sent from a high level Interrupt handler, the source context identifier is set to ADL_CTX_HIGH_LEVEL_IRQ_HANDLER.

Prototype:

s32 adl_msgSend (adl_ctxID_e DestinationTask, u32 MessageIdentifier,u32 Length, void *Data)

Parameters:

DestinationTask: This parameter defines the destination task to which message has to be sent.
MessageIdentifier: This parameter defines the identifier to the message. The message reception filters will be applied to this identifier before notifying the concerned message handlers.
Length: This parameter defines the length of the message. This should be set to 0 if the message doesn't have the message body.
Data: This parameter defines the address of the buffer which has the message body. This should be set to 0 if the message doesn't have the message body.

Returned Values:

OK will be returned on success.
ADL_RET_ERR_PARAM will be returned if a parameter has an incorrect value.

7.6. Sample Code

```
// 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 );
}
```

Summary

The following points have been covered in this chapter:

- **Message service provides the facility to send messages to other tasks.**
- **Following APIs are provided by the Message service:**
    - **adl_msgSubscribe API is used to subscribe for the Message service**
    - **adl_msgUnsubscribe API is used to unsubscribe from the Message service**
    - **adl_msgSend API is used to send the message to the other task**

8. Synchronization – Semaphore APIs

This chapter introduces the synchronization APIs provided by Open AT OS.

8.1. Introduction

Synchronization is a method which allows to co-ordinate multiple paths of execution. For e.g. in a system, you could have a LCD task writing on LCD and another task to get the input data. In this case, LCD task should be synchronized with input task in order to display only when input data is available. Open AT OS provides **Semaphore service** to perform Synchronisation.

Along with Synchronisation, Semaphore can also be used to create **critical section** which cannot be interrupted by any other task. This capability allows controlling the access of multiple tasks to shared resources. Without using semaphores, it is possible for one task to **corrupt** a shared data while another task is in the process of using or updating the data. This often leads to data corruption.

8.2. Synchronization APIs

Open AT OS provides semaphore for synchronizing the resource in Open AT. A semaphore is a protected variable and is a method for restricting access to shared resources (e.g. flash) in a multiprogramming environment. When multiple tasks are in execution, semaphore prevents problems while accessing shared resources among multiple tasks.

For e.g. Let us say, when two tasks wants to write in object id “1” at the same time, it could lead to corruption of flash object. To avoid this, semaphores can be used. Semaphores will make sure that at a time only one task could access the flash object.

The header file adl_sem.h should be included in the Open AT Application to use this service.

To use Semaphore, you can do the following:

- Subscribe to Semaphore with a count value of 1. The count value is equal to number of times a semaphore can be consumed. Here consume means decrementing a semaphore counter value by 1. When the counter value reaches zero, and if a task tries to consume semaphore, the calling task is suspended and resumed only when the semaphore is produced (released) by another task.
- Consume a semaphore before accessing a shared resource for e.g. Flash object.
- Access the shared resource.
- Produce (Release) the semaphore so that other tasks can access the shared resource. When a semaphore is produced, the semaphore counter value is incremented by 1.

8.3. Subscribe to Semaphore Service

This API can be used to subscribe to the semaphore service. The return value of this API should be saved in order to use other Semaphore APIs.

Prototype:

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

Parameters:

SemCounter: This parameter defines the initial value of the semaphore counter.

Returned Values:

A positive semaphore service handle is returned on success.
A negative error value is returned in case of an error.
ADL_RET_ERR_NO_MORE_SEMAPHORES is returned when there are no more free semaphore resources available. There are up to 100 semaphore resources are available.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level Interrupt handler.

8.4. Get Semaphore Resource Count

This API can be used to retrieve the count of currently available free semaphore resources.

Prototype:

```
u32 adl_semGetResourcesCount ( void )
```

Parameters:

None

Returned Values:

This function returns number of free semaphore resource.

8.5. Consume a Semaphore

This API can be used to consume a semaphore. In case Semaphore is consumed already, this task will be suspended till the semaphore is available (Produced).

Prototype:

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

Parameters:

SemHandle: This is the handle returned by adl_semSubscribe API.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned when the supplied handle is unknown.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.

NOTE :
This API could cause an exception 205, if semaphore is consumed too many times.

8.6. Consume a Semaphore with Timeout

This API can be used to consume a semaphore with certain time-out. If semaphore is not available, the API time-outs after defined period.

Prototype:

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

Parameters:

SemHandle: This is the handle returned by adl_semSubscribe API.
TimeOut: This defines the time-out duration in 18.5ms steps.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned when the supplied handle is unknown.
ADL_RET_ERR_PARAM is returned when a supplied parameter value is wrong.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.
ADL_RET_ERR_BAD_STATE is returned, if the semaphore is unavailable after TimeOut.

NOTE : This API could cause an RTK exception 206, if semaphore has been consumed too many times.

8.7. Produce a Semaphore

This API can be used to produce a consumed semaphore.

Prototype:

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

Parameters:

SemHandle: This is the handle returned by adl_semSubscribe API.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned when the supplied handle is unknown.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.

NOTE :
This API could cause an exception 133, if semaphore is produced too many times.

8.8. Unsubscribe from a Semaphore Service

This API can be used to unsubscribe from a semaphore service.

Prototype:

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

Parameters:

SemHandle: This is the handle returned by adl_semSubscribe API.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned when the supplied handle is unknown.
ADL_RET_ERR_BAD_STATE is returned when the semaphore inner counter value is different from the initial value.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.

8.9. Check if a Semaphore is consumed

This API can be used to check if a semaphore is consumed.

Prototype:

```
s32 adl_semIsConsumed ( s32 SemHandle );
```

Parameters:

SemHandle: This is the handle returned by adl_semSubscribe API.

Returned Values:

TRUE, if the semaphore is consumed.
FALSE, if the semaphore is not consumed.
ADL_RET_ERR_UNKNOWN_HDL is returned when the supplied handle is unknown.

8.10. Sample Code

```
// 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 );
}
```

Summary

The following points have been covered in this chapter:

- **A semaphore is a protected variable and is a method for restricting access to shared resources (e.g. flash) in a multiprogramming environment.**
- **Following APIs are provided by the Semaphore service are:**
    - **adl_semSubscribe to subscribe to semaphore service**
    - **adl_semUnsubscribe to unsubscribe from semaphore service**
    - **adl_semConsume to consume a semaphore**
    - **adl_semConsumeDelay to consume a semaphore with time-out**
    - **adl_semProduce to produce a semaphore**
    - **adl_semGetResourceCount to get the number of free semaphore resources**
    - **adl_semIsConsumed to check if semaphore is consumed or not**

9. Thread safe data management – Queue APIs

This chapter introduces the Thread safe data management APIs provided by Open AT OS.

9.1. Introduction

Open AT OS provides the Queue service to manage the thread safe data. Open AT OS makes sure that the data is protected from the concurrent access.

Open AT OS supports both LIFO and FIFO approach for accessing data as shown in the below diagram.

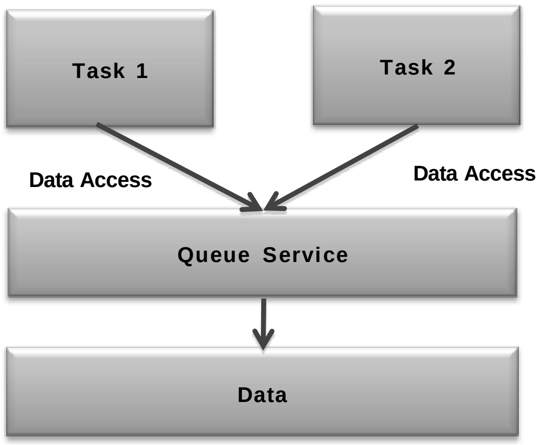


Figure 77 - Thread Safe Data Management Example

9.2. Header file

Open AT OS provides the Queue service to manage the thread safe data. The header file adl_queue.h should be included in Open AT Application to use this service.

9.3. Subscribe to Queue Service

This API can be used to subscribe to the Queue service.

Prototype:

```
s32 adl_queueSubscribe (adl_queueOptions_e Option)
```

Parameters:

Option: This parameter defines the behaviour of the queue resource. This is defined by the adl_queueOptions_e enum as defined below:

```
typedef enum
{
 // First In, First Out method: First pushed item will be retreived first
 ADL_QUEUE_OPT_FIFO,
 // Last In, First Out method: Last pushed item will be retreived first
 ADL_QUEUE_OPT_LIFO,
 //Reserved for internal use
 ADL_QUEUE_OPT_LAST
} adl_queueOptions_e;
```

Returned Values:

A positive queue service handle is returned on success.
ADL_RET_ERR_PARAM is returned on parameter error.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler

9.4. Unsubscribe from Queue Service

This API can be used to unsubscribe from the Queue service.

Prototype:

```
s32 adl_queueUnsubscribe (s32 Handle )
```

Parameters:

Handle: This is the handle returned by the adl_queueSubscribe API.

Returned Values:

A positive queue service handle is returned on success.
ADL_RET_ERR_BAD_STATE is returned if the provided queue resource is not empty.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.
ADL_RET_ERR_UNKNOWN_HANDLE is returned if the provided handle is invalid.

9.5. Check for the Queue State

This API can be used for checking the state of the queue.

Prototype:

```
s32 adl_queueIsEmpty (s32 Handle )
```

Parameters:

Handle: This is the handle returned by the adl_queueSubscribe API.

Returned Values:

FALSE is returned if it remains at least one item in the queue.
TRUE is returned if the queue is empty.
ADL_RET_ERR_UNKNOWN_HANDLE is returned if the provided handle is invalid.

9.6. Adding Item to the Queue

This API can be used to add the item in the queue.

Prototype:

```
s32 adl_queuePushItem (s32 Handle, void* Item )
```

Parameters:

Handle: This is the handle returned by the adl_queueSubscribe API.
Item: This parameter defines the data to be added in the queue. Note that this parameter can't be set to NULL.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HANDLE is returned if the provided handle is invalid.
ADL_RET_ERR_PARAM is returned if the specified parameter is incorrect.

NOTE :
This API could cause an exception 144, if too many items are added in the queue.

9.7. Removing Item from the Queue

This API can be used to retrieve the item from the queue. The item will be removed after retrieval.

Prototype:

```
void* adl_queuePopItem (s32 Handle )
```

Parameters:

Handle: This is the handle returned by the adl_queueSubscribe API.

Returned Values:

A pointer to the item is returned on success.
NULL is returned, if the queue is empty or the handle is unknown.

9.8. Sample Code

```
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 );
}
```

Summary

The following points have been covered in this chapter:

- **A Queue service is provided for providing thread safe data management.**
- **This serviced can be used either by FIFO or LIFO method.**
- **Following APIs are provided by the Queue service are:**
    - **adl_queueSubscribe to create a queue with LIFO or FIFO approach**
    - **adl_queueUnsubscribe to unsubscribe from queue service**
    - **adl_queueIsEmpty to check if the queue is empty**
    - **adl_queuePushItem to add item to the queue**
    - **adl_queuePopItem to retrieve item from the queue**

10. Interrupt Handling – IRQ Service

10.1. Introduction

Interrupts are asynchronous breaks in program flow that occur as a result of events outside the running program. They are usually hardware related, occurring due to events such as a button press, timer expiration or completion of a data transfer.

RTOS feature of the Open AT OS help you to handle interrupts generated by embedded module:

- IRQ service to manage all the interrupts
- External Interrupt Service for handling external interrupts
- TCU Service for handling timer interrupts
- Audio Service for handling audio interrupts

10.2. IRQ Overview

RTOS feature of the Open AT OS help you to handle interrupts generated by embedded module. IRQ service provides a handler function, which is invoked when an interrupt is generated. IRQ service manages the interrupt generated through following sources:
External interrupt generated through the external interrupt pin provided on embedded module.

- Interrupts generated by the TCU service.
- Interrupts generated by the Audio service.

To use an interrupt, you can do the following:
Subscribe to handler using IRQ service and save the return value (handle) in a variable.
Attach this handle to an interrupt service such as External interrupt or TCU service or Audio service.
When the interrupt occurs, the handler defined using IRQ service is invoked by Open AT OS.

IRQ service provides the facility to define the notification level of the interrupt handler during subscription. The handler can be defined either as high-level handler or as a low-level handler. Lower the level of the handler faster its notification.

10.3. Interruption call stack size definition

The interruption handlers will run from the Open AT Application context, either directly in an interruption context (for Low-level handlers), or in a specific Interruption processing Task context (for High-level handlers). Since the interrupt handler run in different contexts, a separate call stack is defined for each one of these contexts (direct interruption, and Interruption Processing Task ones).

Open AT Application task, call stack size is defined through constants. If the application does not provide one of them, it will automatically be set to zero, and the associated context (Low level or High level) will not be usable at run time.

The size of the stack is determined using the following:

- Size of the local variables defined in each calling function
- Number of parameters a function can accept
- Number of function calls done in recursion

During run-time, if the stack size is insufficient, Open AT OS causes an exception and resets the embedded module.

Prototype:

```
const u32 adl_InitIRQLowLevelStackSize =  <size>
const u32 adl_InitIRQHighLevelStackSize =  <size>
```

Parameters:

Size: Size of the stack.

10.4. Interrupt handler processing

Low-level interrupt handler gives you the access to execute the time critical process. Lower level handlers are given higher priority in the Open AT Application. Their priority is higher than Open AT Application, AT command task, Firmware (GSM task) but lesser than Firmware interrupt (GSM interrupt). 1ms is the maximum reaction time for the lower level interrupt handler.
Below diagram shows the processing of the various tasks in the Open AT OS.

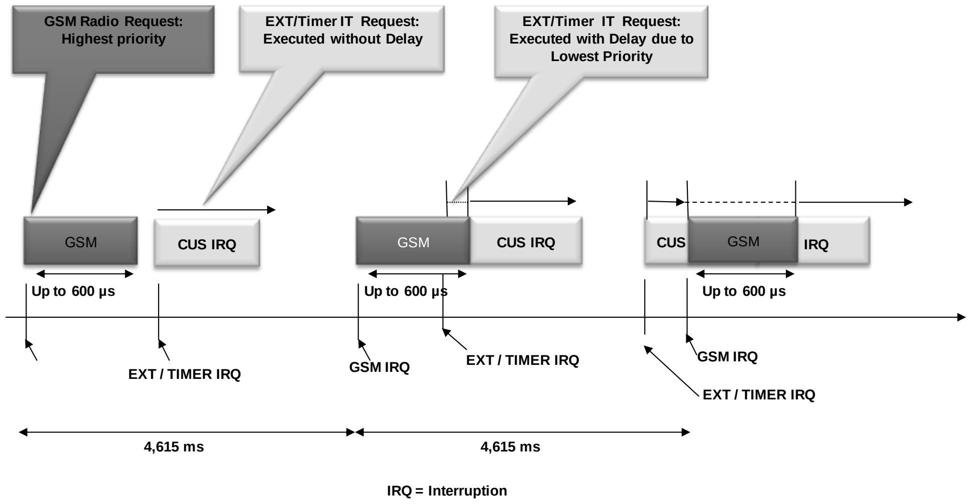


Figure 78 - External Interrupt Processing With Level 1 Interrupt

As shown in the above figure, the GSM interrupt has more priority than the Level 1 Open AT interrupt. Hence when Open AT Interrupt is in execution, the GSM interrupt can preempt the execution.

10.5. Header file

The ADL IRQ service allows interrupt handlers to be defined. These handlers are usable with other services (External Interrupt Pins, Audio, Timers) to monitor specific interrupt sources.
The header file adl_irq.h should be included to use the IRQ service APIs.

NOTE :
The Real Time Enhancement feature has to be enabled on the embedded module in order to use this service.
AT+WCFM=5 command can be used to check the Real Time Enhancement feature availability.

10.6. Subscribe to IRQ service

This API can be used to subscribe to the IRQ service. This handler is used to monitor the interrupt generated by other services such as TCU or audio or external interrupt.

Prototype:

s32 adl_irqSubscribe (adl_irqHandler_f IrqHandler, adl_irqNotificationLevel_e NotificationLevel, adl_irqPriorityLevel_e PriorityLevel, adl_irqOptions_e Options)

Parameters:

IrqHandler: Call back function defined by the application. This handler will be called when the interrupt is triggered.

Prototype:

bool (*adl_irqHandler_f) (adl_irqID_e Source, adl_irqNotifyLevel_e NotificationLevel, adl_irqEventData_t *Data);)

Parameters:

Source: Source of the interrupt. These are defined in the enum given below:

```
typedef enum
{
  //RX path interrupt sources raised by the Audio Stream Listen
  //service
  ADL_IRQ_ID_AUDIO_RX_LISTEN,
  // TX path interrupt sources raised by the Audio Stream Listen
  //service
  ADL_IRQ_ID_AUDIO_TX_LISTEN,
  // RX path interrupt sources raised by the Audio Stream Play
  //service
  ADL_IRQ_ID_AUDIO_RX_PLAY,
  // TX path interrupt sources raised by the Audio Stream Play
  //service
  ADL_IRQ_ID_AUDIO_TX_PLAY,
  // Interrupt sources raised by the External Interrupt Pin source
  ADL_IRQ_ID_EXTINT,
  // Interrupt sources raised by the Timer Interrupts source
  ADL_IRQ_ID_TIMER,
  // Capture interrupt sources raised by the Timer Interrupts source
  ADL_IRQ_ID_EVENT_CAPTURE,
  // Detection interrupt sources raised by the Timer Interrupt source
  ADL_IRQ_ID_EVENT_DETECTION,
  // SPI bus asynchronous end of transmission event
  ADL_IRQ_ID_SPI_EOT,
  // I2C bus asynchronous end of transmission event
  ADL_IRQ_ID_I2C_EOT,
  // Reserved for internal use
  ADL_IRQ_ID_LAST
} adl_irqID_e;
```

NotificationLevel: Level of notification. These are defined in the enum given below:

```
typedef enum
{
  ADL_IRQ_NOTIFY_LOW_LEVEL, // Low level interrupt handlers
  ADL_IRQ_NOTIFY_HIGH_LEVEL,  // High level interrupt handlers
  ADL_IRQ_NOTIFY_LAST // Reserved for internal use
} adl_irqNotificationLevel_e;
```

Data: Input/output data field is specific to the type of the interrupt. These are defined in the structure given below:

```
typedef struct
{
  union
  {
    void * LowLevelOuput;
    void * HighLevelInput;
  } UserData;  //Allows the application to exchange data between
                 //low level and high level interrupt handlers
  void * SourceData; //Provides to handlers an interrupt source
                        //specific data
  u32 Instance //Instance identifier of the interrupt event which has
                //just occurred.
  void * Context //Context
} adl_irqEventData_t;
```

Returned Values:
Return value is valid only valid in case of low-level handle.
TRUE: The higher-level interrupt will be called if it is subscribed for the same interrupt.
FALSE: The higher-level interrupt corresponding to the interrupt will not be called.
NotificationLevel: Allow to categorize a handler as a low level or a high-level handler. Lower the level of the handler faster is the notification. The different level of the handler that can be set is defined in the following enum.

```
typedef enum
{
  ADL_IRQ_NOTIFY_LOW_LEVEL, // Low level interrupt handlers
  ADL_IRQ_NOTIFY_HIGH_LEVEL,  // High level interrupt handlers
  ADL_IRQ_NOTIFY_LAST // Reserved for internal use
} adl_irqNotificationLevel_e;
```

Parameters:
ADL_IRQ_NOTIFY_LOW_LEVEL: The value defines the level of notification of the handler as low. This handler has priority higher than Firmware task, AT command task, Open AT Application task and the higher-level handler but lower than Firmware interrupt. The maximum time of execution of this type of handler is 1ms.
ADL_IRQ_NOTIFY_HIGH_LEVEL: The value defines the level of notification of the handler as high. The handler with this level will have a priority higher that AT command task, Open AT Application task but lower than Firmware interrupt Firmware task and lower level handler. The maximum time of execution of this type of handler is 1ms – 10ms.
PriorityLevel: This parameter defines the priority level of the given interrupt handler.
The lowest priority level is always 0.
The highest priority level should be retrieved using adl_irqGetCapabilities API.
Options: This parameter defines the interrupt handler notification options. This parameter can have the following value:
ADL_IRQ_OPTION_AUTO_READ: If this option is set, than source related information structure is automatically read by IRQ service and supplied to the interruption handler. In case of higher level handler, this is the only way to retrieve the source information.

Returned Values:

A positive or null IRQ service handle is returned on success. This handle will be used further IRQ & interrupt source services function calls.
ADL_RET_ERR_PARAM is returned, if the parameters specified are incorrect.
ADL_RET_ERR_NOT_SUBSCRIBED is returned, if a low or high level handler subscription is required while the associated context call stack size was not supplied by the application.
ADL_RET_ERR_BAD_STATE is returned if the function is called in RTE mode.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler (the function is forbidden in this context)

NOTE :
This API cannot be used in RTE mode. It returns ADL_RET_ERR_BAD_STATE in case if it is used in RTE mode.

10.7. Subscribe to IRQ service using additional options

This API can be used to subscribe to the IRQ service. This API is same as adl_irqSubscribe() except that this API allows to define all the parameters through a structure instead of independent configurations.

Prototype:

s32 adl_irqSubscribeExt (adl_irqHandler_f IrqHandler, adl_irqNotificationLevel_e NotificationLevel, adl_irqConfig_t* Config)

Parameters:

IrqHandler: Call back handler defined by the application. This handler will be called when the interrupt is triggered.

Prototype:

```
bool (*adl_irqHandler_f) (adl_irqID_e Source, adl_irqNotifyLevel_e NotificationLevel,
adl_irqEventData_t *Data);)
```

Parameters:

Source: Source of the interrupt. These are defined in the enum given below:

```
typedef enum
{
  //RX path interrupt sources raised by the Audio Stream Listen
  //service
  ADL_IRQ_ID_AUDIO_RX_LISTEN,
  // TX path interrupt sources raised by the Audio Stream Listen
  //service
  ADL_IRQ_ID_AUDIO_TX_LISTEN,
  // RX path interrupt sources raised by the Audio Stream Play
  //service
  ADL_IRQ_ID_AUDIO_RX_PLAY,
  // TX path interrupt sources raised by the Audio Stream Play
  //service
  ADL_IRQ_ID_AUDIO_TX_PLAY,
  // Interrupt sources raised by the External Interrupt Pin source
  ADL_IRQ_ID_EXTINT,
  // Interrupt sources raised by the Timer Interrupts source
  ADL_IRQ_ID_TIMER,
  // Capture interrupt sources raised by the Timer Interrupts source
  ADL_IRQ_ID_EVENT_CAPTURE,
  // Detection interrupt sources raised by the Timer Interrupt source
  ADL_IRQ_ID_EVENT_DETECTION,
  // SPI bus asynchronous end of transmission event
  ADL_IRQ_ID_SPI_EOT,
  // I2C bus asynchronous end of transmission event
  ADL_IRQ_ID_I2C_EOT,
  // Reserved for internal use
  ADL_IRQ_ID_LAST
} adl_irqID_e;
```

NotificationLevel: Level of notification. These are defined in the enum given below:

```
typedef enum
```

```
{
  ADL_IRQ_NOTIFY_LOW_LEVEL, // Low level interrupt handlers
  ADL_IRQ_NOTIFY_HIGH_LEVEL,  // High level interrupt handlers
  ADL_IRQ_NOTIFY_LAST // Reserved for internal use
} adl_irqNotificationLevel_e;
```

Data: Input/output data field is specific to the type of the interrupt. These are defined in the structure given below:

```
typedef struct
{
  union
  {
    void * LowLevelOuput;
    void * HighLevelInput;
  } UserData;  //Allows the application to exchange data between
                 //low level and high level interrupt handlers
  void * SourceData; //Provides to handlers an interrupt source
                         //specific data
  u32 Instance //Instance identifier of the interrupt event which has
                //just occurred.
  void * Context //Context
} adl_irqEventData_t;
```

Returned Values:

Return value is valid only valid in case of low-level handle.

TRUE: The higher-level interrupt will be called if it is subscribed for the same interrupt.
FALSE: The higher-level interrupt corresponding to the interrupt will not be called.
NotificationLevel: Allow to categorize a handler as a low level or a high-level handler. Lower the level of the handler faster is the notification. The different level of the handler that can be set is defined in the following enum.

```
typedef enum
{
  ADL_IRQ_NOTIFY_LOW_LEVEL, // Low level interrupt handlers
  ADL_IRQ_NOTIFY_HIGH_LEVEL,  // High level interrupt handlers
  ADL_IRQ_NOTIFY_LAST // Reserved for internal use
} adl_irqNotificationLevel_e;
```

Parameters:

ADL_IRQ_NOTIFY_LOW_LEVEL: The value defines the level of notification of the handler as low. This handler has priority higher than Firmware task, AT command task, Open AT Application task and the higher-level handler but lower than Firmware interrupt. The maximum time of execution of this type of handler is 1ms.

ADL_IRQ_NOTIFY_HIGH_LEVEL: The value defines the level of notification of the handler as high. The handler with this level will have a priority higher that AT command task, Open AT Application task but lower than Firmware interrupt Firmware task and lower level handler. The maximum time of execution of this type of handler is 1ms – 10ms.

Config: This parameter defines the interrupt handler configuration. This is defined by the adl_irqConfig_t structure as mentioned below:

```
typedef struct
{
 adl_irqPriorityLevel_e PriorityLevel,
 bool Enable,
 u8 Pad[2] // Reserved for future use
 adl_irqOptions_e Options
} adl_irqConfig_t;
```

Parameters:

PriorityLevel: This parameter defines the interrupt handler priority.

The lowest priority level is always 0.

The highest priority level should be retrieved using adl_irqGetCapabilities API.

Enable: This parameter field defines if the interrupt handler is enabled or not.

TRUE: If it is set to TRUE, the interrupt handler is enabled and any interrupt event on which is plugged this handler will call the related function.

FALSE: If it is 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.

Options: This parameter defines the interrupt handler notification options. This parameter can have the following value:

ADL_IRQ_OPTION_AUTO_READ: If this option is set, than source related information structure is automatically read by IRQ service and supplied to the interruption handler. In case of higher level handler, this is the only way to retrieve the source information.

Returned Values:

A positive or null IRQ service handle is returned on success. This handle will be used further IRQ & interrupt source services function calls.

ADL_RET_ERR_PARAM is returned, if the parameters are incorrect.

ADL_RET_ERR_NOT_SUBSCRIBED is returned, if a low or high level handler subscription is required while the associated context call stack size was not supplied by the application.

ADL_RET_ERR_BAD_STATE is returned, if the function is called in RTE mode.

ADL_RET_ERR_SERVICE_LOCKED is returned, if the function was called from a low level interrupt handler (the function is forbidden in this context)

NOTE :
This API cannot be used in RTE mode. It returns ADL_RET_ERR_BAD_STATE in case if it is used in RTE mode.

10.8. Unsubscribe from IRQ Service

This API should be used to unsubscribe from the IRQ service.

Prototype:

```
s32 adl_irqUnsubscribe ( s32 IrqHandle )
```

Parameters:

IrqHandle: This parameter is the IRQ handle returned during subscription for IRQ service.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned, if the supplied handle is unknown.
ADL_RET_ERR_BAD_STATE is returned, if the supplied handle is still used by an interrupt source service.
ADL_RET_ERR_SERVICE_LOCKED is returned, if the function was called from a low level interrupt handler.

10.9. Configure Interrupt Handler Parameters

This API can be used to modify the interrupt handler configurations set using adl_irqSubscribeExt API.

Prototype:

```
s32 adl_irqSetConfig ( s32 IrqHandle, adl_irqConfig_t * Config )
```

Parameters:

IrqHandle: This parameter is the IRQ handle returned during subscription for IRQ service.
Config: This parameter is the configuration structure which defines the configuration parameters of the interrupt handler.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the supplied handle is unknown.
ADL_RET_ERR_PARAM is returned, if the parameters are incorrect.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.

10.10. Get the Interrupt Handler Parameters

This API can be used to retrieve the current configurations of the interrupt handler.

Prototype:

```
s32 adl_irqGetConfig ( s32 IrqHandle, adl_irqConfig_t * Config )
```

Parameters:

IrqHandle: This parameter is the IRQ handle returned during subscription for IRQ service.
Config: This parameter is the configuration structure which holds the configuration parameters retrieved using this API.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the supplied handle is unknown.
ADL_RET_ERR_PARAM is returned if the parameters are incorrect.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.

10.11. Get the Interrupt Service Capabilities

This API can be used to retrieve the interrupt service capabilities on the current platform.

Prototype:

```
s32 adl_irqGetCapabilities ( adl_irqCapabilities_t * Capabilities )
```

Parameters:

Capabilities: This parameter defines the IRQ service capabilities. This is defined by the below structure:

```
typedef struct
{
  u8 PriorityLevelsCount, // Priority level count
  u8 Pad [3] // Reserved for internal use
  u8 InstancesCount [ADL_IRQ_ID_LAST] // Instance count for each interrupt
                                      //source identifier
} adl_irqCapabilities_t;
```

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if there the parameter specified is incorrect.

10.12. Sample Code

```
// 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
.. // Highest priority
 Config.PriorityLevel = Caps.PriorityLevelsCount - 1;
 Config.Enable = TRUE; // Interrupt handler enabled
 // Auto-read option set
 Config.Options = ADL_IRQ_OPTION_AUTO_READ;
 // 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 );
}
```

Summary

The following points have been covered in this chapter:

- **IRQ service provides the facility to subscribe to the interrupt handler.**
- **These handlers will be called when the service linked to them trigger as interrupt.**
- **Interrupt handler can be linked to the following:**
    - **External interrupt**
    - **TCU timer**
    - **SPI and I2C buses**
    - **Audio streaming process**
- **In API provided by the IRQ service are:**
    - **adl_irqSubscribe/adl_irqSubscribeExt API is used to subscribe for the IRQ service**
    - **adl_irqUnsubscribe API is used to unsubscribe from the IRQ service**
    - **adl_irqSetConfig API is used to configure interrupt handler configurations**
    - **adl_irqGetConfig API is used to retrieve the current configuration of interrupt handler**
    - **adl_irqGetCapabilities API is used to retrieve the capabilities of the IRQ service on the current platform**
- **The handlers subscribed are categorize as higher or lower level handler.**

1. **Lower the level of the handler faster is its notification.**

11. External Interrupt Service

11.1. Objective

This chapter will introduce you to external interrupts provided on the embedded module and how to use them in your Open AT Application.

11.2. Introduction

Embedded module is provided with multiple external interrupts. These interrupts can be used in the Open AT Application. The priority of handlers depends on the level of the handler defined at the time of subscription of the handler.
The external interrupts are generated either on:

- the falling or the rising edge of the input signal, or both
- the low or high level of the input signal

External interrupts are first processed by a filter and then interrupt handlers are called as shown in the below diagram.

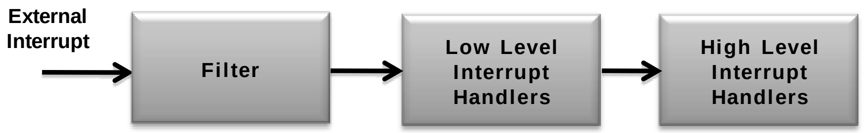


Figure 79- External Interrupt Management

Open AT OS supports following filter configurations:
Bypass (No Filter): Interrupts are generated immediately and there will not be any delay in the generation of interrupts.

Debounce: Stable signal is required to generate the interrupts. Hence there will be delay in the generation of interrupts as shown in the below diagram.

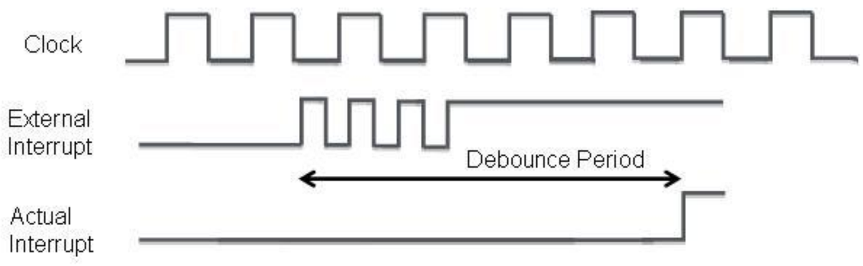


Figure 80 - Debounce

Stretching: Signal is stretched to stretched in order to detect even small glitches in the signal as shown in the below diagram.

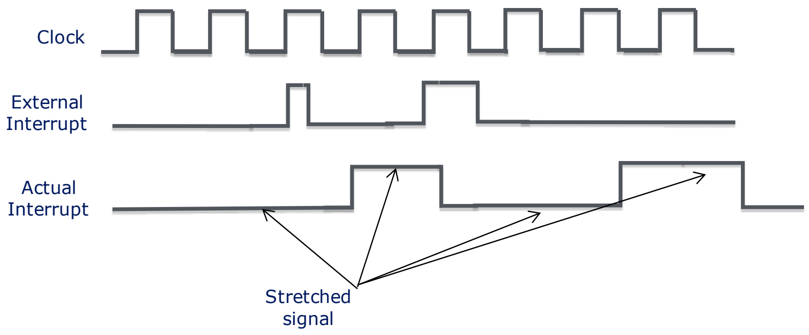


Figure 81 - Stretching

11.3. Header file

The header file to be included for using External interrupt APIs in Open AT Application is adl_extint.h.

11.4.Subscribe to External Interrupt

This API is used to subscribe to the external interrupt. This API will link the external interrupt with the handler provided by the IRQ service.

Prototype:

```
s32 adl_extintSubscribe (adl_extintID_e ExtIntID, s32 LowLevelIrqHandle, s32 HighLevelIrqHandle, adl_extintConfig_t * Settings )
```

Parameters:

ExtIntID: Identifier of the external interrupt to be subscribed. These identifiers are defined in the adl_extintID_e enum.

LowLevelIrqHandle: This parameter is the IRQ handle returned by IRQ service for low level interrupts.
HighLevelIrqHandle: This parameter is the IRQ handle returned by IRQ service for high level interrupts.
Settings: The configuration for the external interrupt. These settings are defined in the adl_extintConfig_t structure.

Returned Values:

A positive or null value is returned on success.
A negative error value is returned on error:
ADL_RET_ERR_PARAM is returned if one parameter has an incorrect value.
ADL_RET_ERR_NOT_SUPPORTED is returned if one parameter refers to a mode or a configuration not supported by the embedded module.
ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the service was already subscribed for this external interrupt pin.
ADL_RET_ERR_BAD_HDL is returned if one or both supplied interrupt handler identifiers are invalid.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

11.5. Configure the External Interrupt

This API is used to change the configuration of the external interrupt.

Prototype:

```
s32 adl_extintConfig ( s32 ExtIntHandle, adl_extintConfig_t * Settings )
```

Parameters:

ExtIntHandle: Handle returned during external interrupt service subscription.
Settings: This defines the external pin configuration.

Returned Values:

OK is returned on success.
A negative error value is returned on error:
ADL_RET_ERR_PARAM is returned if one parameter has an incorrect value.
ADL_RET_ERR_NOT_SUPPORTED is returned if one parameter refers to a mode or a configuration not supported by the embedded module.
ADL_RET_ERR_UNKNOWN_HDL is returned if the supplied External Interrupt handle is unknown.

11.6. Read the External Interrupt Configuration

This API is used to read the external interrupt configuration.

Prototype:

```
s32 adl_extintGetConfig ( s32 ExtIntHandle, adl_extintConfig_t * Settings )
```

Parameters:

ExtIntHandle: Handle returned during external interrupt service subscription.
Settings: External pin configuration to be retrieved.

Returned Values:

OK is returned on success.
A negative error value is returned on error:
ADL_RET_ERR_PARAM is returned if one parameter has an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the supplied External Interrupt handle is unknown.

11.7. Read the External Interrupt

This API is used to read the status of the external interrupt.

Prototype:

```
s32 adl_extintRead (s32 ExtIntHandle, adl_extintInfo_t * Info )
```

Parameters:

ExtIntHandle: This is the handle returned during subscription.
Info: This is the external pin information as defined by the strcture adl_extintInfo_t.

```
typedef struct
{
  u8 PinState; // External interrupt pin status
} adl_extintInfo_t;
```

Returned Values:

OK is returned on success.
A negative error value is returned on error:
ADL_RET_ERR_PARAM is returned if one parameter has an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the supplied External Interrupt handle is unknown.

11.8. Unsubscribe from the External Interrupt

This API is used to unsubscribe from the external interrupt service.

Prototype:

```
s32 adl_extintUnsubscribe (s32 ExtIntHandle )
```

Parameters:

ExtIntHandle: This is the handle returned during subscription.

Returned Values:

OK is returned on success.
A negative error value is returned on error:
ADL_RET_ERR_UNKNOWN_HDL is returned if the supplied External Interrupt handle is unknown
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

11.9. Get the capabilities of the External Interrupt

This API is used to get the capabilities of the external interrupt.

Prototype:

```
s32 adl_extintGetCapabilities (adl_extintCapabilities_t * PinCapabilities)
```

Parameters:

PinCapabilities: This is the pointer to the adl_extintCapablities_t structure which holds the capabilities of the external interrupt.

```
typedef struct
{
  u8 NbExternalInterrupt; //Number of external interrupt
  bool RisingEdgeSensitivity //Rising edge sensitivity supported
  bool FallingEdgeSensitivity //Falling edge sensitivity supported.
  bool BothEdgeSensitivity //Both edge detector supported.
  bool LowLevelSensitivity //Low level sensitivity supported.
  bool HighLevelSensitivity //High level sensitivity supported.
  bool BypassMode //Bypass mode supported.
  bool StretchingMode //Stretching mode supported.
  bool DebounceMode //Debounce mode supported.
  u8 MaxDebounceDuration //Debounce max duration in ms.
  u8 DebounceNbStep //Number of step for debounce duration.
  u8 PriorityLevelsCount //Available priority levels for the External Interrupt
                         //service (to be used as a adl_irqPriorityLevel_e
                          // value in the IRQ service
  u8 Pad [4] //Internal use.
} adl_extintCapabilities_t
```

Returned Values:

OK is returned on success.
A negative error value is returned on error:
ADL_RET_ERR_PARAM is returned if one parameter has an incorrect value.

11.10. Set FIQ status

This API is used to set the FIQ status. Set to TRUE for enabling and FALSE to disable 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: Status to be set.

Returned Values:

OK is returned on success.
A negative error value is returned on error:
ADL_RET_ERR_PARAM if the parameter has 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

11.11. Get FIQ status

This API is used to get 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:

OK is returned on success.
A negative error value is returned on error:
ADL_RET_ERR_PARAM if the parameter has incorrect value
ADL_RET_ERR_UNKNOWN_HDL if the supplied external interruption handle is unknown

11.12. Sample Code

```
// 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 interrupt handler
bool MyExtIntHandler ( adl_irqID_e Source, adl_irqNotifyLevel_e
NotificationLevel,adl_irqEventData_t * Data )
{
  // Read the input status
  adl_extintInfo_t Status, * AutoReadStatus;
  adl_extintRead ( ExtIntHandle, &Status );
  // Input status can also be obtained from the auto read option.
  AutoReadStatus = ( adl_extintInfo_t * ) Data->SourceData;
  return TRUE;
}
// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
  adl_extintCapabilities_t My_ExtInt_Capa;
  adl_extintGetCapabilities ( &My_ExtInt_Capa );
  // Test if the embedded module 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 );
  }
}
void MyFunction2 ( void )
```

```
            {
              // Un-subscribes from the ExtInt service
              adl_extintUnsubscribe ( ExtIntHandle );
            }
```

Summary

The following points have been covered in this chapter:

- **External Interrupt service provides the facility to service the interrupts generated by the external interrupt pins.**
- **The API provided by the External Interrupt service are:**
    - **adl_extintSubscribe API is used to subscribe for the External interrupt service**
    - **adl_extintUnsubscribe API is used to unsubscribe from the External Interrupt service**
    - **adl_extintSetConfig API is used to configure external interrupt configurations**
    - **adl_extintGetConfig API is used to retrieve the current configuration of external interrupt**
    - **adl_extintRead API is used to read the status of the external interrupt**
    - **adl_extintGetCapabilities API is used to retrieve the capabilities of the external interrupts**

12. Timer and Capture Unit Service

12.1.Objective

This chapter will introduce you to TCU service provided on the embedded module to handle the interrupts generated by the timers.

12.2. Introduction

The TCU service serves the following needs of a real time system:
Provides a way to define accurate timers with a precision in order of µs.
Allows to measure the number of edges (Events) occurred on a PIN in a given stipulated time.
Allows detecting an edge in a given time.

There are three types of TCU service which are provided by Open AT OS as mentioned below:

- Accurate Timers Service
- Event Capture Service
- Event Detection Service

12.3. Accurate Timers Service

This service is used to generate accurate timer events with a precision of:

- Micro-seconds
- Milli-seconds
- Seconds

The timer duration is configured using adl_tcuSubscribe API. When the timer expires, the interrupt handler defined by the IRQ service will be notified with the following parameters:

- 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

12.4. Event Capture Service

This service is used to count the number of events occurred in a given time period on a given embedded module pin. The event count duration is configured using adl_tcuSubscribe API. When the event count duration expires, the interrupt handler defined by the IRQ service will be notified with the following parameters:

- 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 occurred 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

This service provides an option to either specify time period or not.
If the time period is not mentioned, a variable can be used to store the number of events occurred so far.
If the time period is mentioned, interrupt events will be generated on timer expiry.

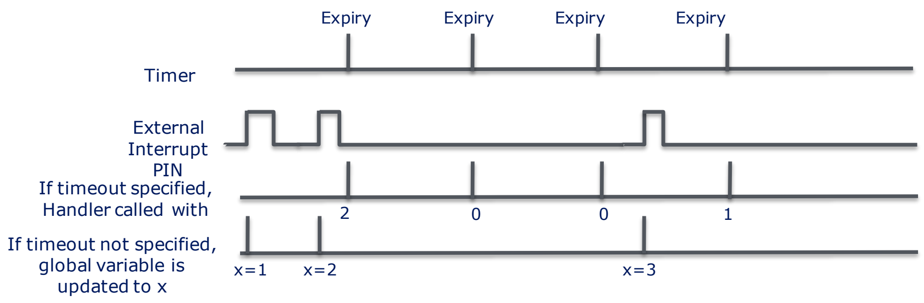

Expiry
Expiry
Expiry
Expiry
Timer
External Interrupt PIN
If timeout specified, Handler called with
2
0
0
1
If timeout not specified, global variable is updated to x
x=1
x=2
x=3

Figure 82 - Event Capture Service

12.5. Event Detection Service

This service is used to detect events on a given embedded module pin. The interrupt handler defined by the IRQ service will be notified:
if no event is detected in a given period of time configured by adl_tcuSubscribe API, the interrupt handler is called to inform the inactivity
if embedded module pin is interrupted

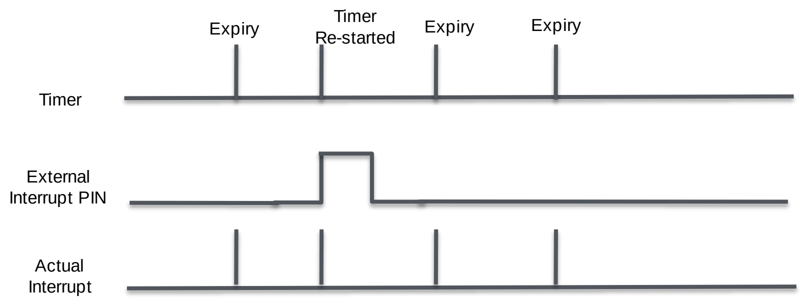


Figure 83 - Event Detection Service

The interrupt handler will be notified with the following parameters:

- 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 cast 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.

12.6. Header file

Open AT OS provides TCU Service interface to handle operations related to the embedded module hardware timers & capture units. Open AT Application should include adl_tch.h header file to use TCU service.

12.7. Subscribe to TCU Service

This API is used to subscribe to the TCU service.

Prototype:

```
s32 adl_tcuSubscribe ( adl_tcuService_e SrvID, s32 LowLevelIrqHandle, s32 HighLevelIrqHandle, void * Settings, void * Context )
```

Parameters:

SrvID: This parameter defines the type of TCU service to be subscribed. This is defined by the adl_tcuService_e enum as mentioned below:

```
enum
{
 ADL_TCU_ACCURATE_TIMER, //Accurate timer service
 ADL_TCU_EVENT_CAPTURE,  //Event capture service
 ADL_TCU_EVENT_DETECTION  //Event detection service
} adl_tcuService_e;
```

LowLevelIrqHandle: This parameter is the IRQ handle returned by IRQ service for low level interrupts.

HighLevelIrqHandle: This parameter is the IRQ handle returned by IRQ service for high level interrupts.

Settings: This parameter is the TCU service configuration according to the SrvID parameter value.

Accurate Timer Service Settings: Following structure defines the parameter settings which should be used for configuring accurate timer:

```
typedef struct
{
 adl_tcuTimerDuration_t Duration,
 u32 Periodic
} adl_tcuTimerSettings_t;
```

Parameters:
Duration: This parameter defines the time duration used for defining accurate timers. This is defined by the adl_tcuTimerDuration_t structure as mentioned below:

```
typedef struct
{
 u32 DurationValue,
 adl_tcuTimerUnit_e DurationUnit
} adl_tcuTimerDuration_t;
```

Parameters:
Duration Value: This parameter defines the timer duration value set by the DurationUnit parameter.
Duration Unit: This parameter defines the unit of the timer duration. This is defined by the following enumeration:

```
typedef enum
{
 //Timer duration is in microseconds
 ADL_TCU_TIMER_UNIT_US = 1,
 // Timer duration is in milliseconds
 ADL_TCU_TIMER_UNIT_MS = 1000,
 // Timer duration is in seconds
 ADL_TCU_TIMER_UNIT_S = 100000,
 //Reserved for future use
 ADL_TCU_TIMER_UNIT_ALIGN = 0x7FFFFFFF
} adl_tcuTimerUnit_e;
```

Periodic: This parameter defines whether it is a periodic timer or not.
TRUE: If it is set to TRUE, the timer is reloaded after each timer event occurrence (timer expiry).
FALSE: If it is set to FALSE, the timer is stopped after the first event occurrence (timer expiry).

Event Capture Service Settings: Following structure defines the parameter settings which should be used for configuring event capture service:

```
typedef struct
{
 u16 CapturePinID
 adl_tcuEventType_e EventType
 u32 Duration
 u32* EventCounter
} adl_tcuEventCaptureSettings_t;
```

Parameters:
CapturePinID: This parameter is the pin identifier on which the service has to monitor events.
EventType: This parameter is the type of events as mentioned below:

```
enum
{
 // No event detected
 ADL_TCU_EVENT_TYPE_NONE = (s16)0xFFFF,
 // Capture or detect rising edge events only
 ADL_TCU_EVENT_TYPE_RISING_EDGE = 0,
 // Capture or detect falling edge events only
 ADL_TCU_EVENT_TYPE_FALLING_EDGE,
 // Capture or detect events on both edges
 ADL_TCU_EVENT_TYPE_BOTH_EDGE
} adl_tcuEventType_e;
```

Duration: This parameter defines the duration of the capture. This is used only if the **EventCounter** parameter is set to NULL, otherwise it will be ignored. When this parameter is used, the related IRQ service handlers are called on every duration expiry, indicating to the application the number of events that have occurred since the previous handler call.

EventCounter: This parameter is an address of 32 bit variable provided by the application. Note that the address variable has to either a global/static one, or an allocated heap buffer. This is used to store the events counter value.
If this address is provided, interrupt events will not be generated, but the event counter value will be incremented, each time a new event is detected.
If this address is set to NULL, the service will generate events, on the time base defined by the **Duration** parameter.

Event Detection Service Settings: Following structure defines the parameter settings which should be used for configuring event detection service

```
typedef struct
{
 u16 DetectionPinID,
 adl_tcuEventType_e EventType,
 u32 Duration
} adl_tcuEventDetectionSettings_t;
```

Parameters:
DetectionPinID: This parameter is the pin identifier on which the service has to monitor events.
EventType: This parameter is the type of events as mentioned below:

```
enum
{
 // No event detected
 ADL_TCU_EVENT_TYPE_NONE = (s16)0xFFFF,
 // Capture or detect rising edge events only
 ADL_TCU_EVENT_TYPE_RISING_EDGE = 0,
 // Capture or detect falling edge events only
 ADL_TCU_EVENT_TYPE_FALLING_EDGE,
 // Capture or detect events on both edges
 ADL_TCU_EVENT_TYPE_BOTH_EDGE
} adl_tcuEventType_e;
```

Duration: This parameter is a optional parameter used to determine the duration for which there has been no edge/pulse on a interrupt pin and generates event to notify handler that no event has occurred for a given time slot.
If it is set to 0, the inactivity detection will be disabled.
If this value is greater than 0, it is the inactivity detection period duration: if no event has occurred since the last notification when the duration expires, the associated handlers will be called to warn the application about this inactivity.
Context: This parameter is the application context, which will be provided back to the application when the TCU events occur.

Returned Values:

A positive TCU service handle is returned on success. This handle is in further TCU service function calls.
ADL_RET_ERR_PARAM is returned on a supplied parameter error.
ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the service was already subscribed for this configuration.
ADL_RET_ERR_BAD_HDL is returned if one or both supplied interrupt handler identifiers are invalid.
ADL_RET_ERR_BAD_STATE is returned if the function is called in RTE mode.
ADL_RET_ERR_NOT_SUPPORTED is returned if the required service is not supported on the current platform.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function was called from a low level interrupt handler.

NOTE :
This API cannot be used in RTE mode. It returns ADL_RET_ERR_BAD_STATE in case if it is used in RTE mode.

12.8. Unsubscribe from TCU Service

This API is used to unsubscribe from the TCU service.

Prototype:

```
s32 adl_tcuUnsubscribe ( s32 Handle )
```

Parameters:

Handle: This parameter is the TCU service handle returned by adl_tcuSubscribe API.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the TCU handle is unknown.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler

12.9. Start the TCU Service

This API is used to start the TCU service event generation. Once the service is started, interrupt events are generated according to the service configuration set using adl_tcuSubscribe API.

Prototype:

```
s32 adl_tcuStart ( s32 Handle )
```

Parameters:

Handle: This parameter is the TCU service handle returned by adl_tcuSubscribe API.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the TCU handle is unknown.

12.10. Stop the TCU Service

This API is used to stop the TCU service event generation. Once the service is stopped interrupt events will not be generated anymore.

Prototype:

```
s32 adl_tcuStop ( s32 Handle, void* OutParam )
```

Parameters:

Handle: This parameter is the TCU service handle returned by adl_tcuSubscribe API.
OutParam: This is the optional output parameter based on the service type. If not used, it should be set to NULL.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the TCU handle is unknown.

NOTE :
If the TCU service is already stopped, calling this API will not any effect. In this scenario it returns OK as response.

12.11. Sample Code

Accurate Timer Service:

```
#include "adl_global.h"
#include “adl_tcu.h”
const u16 wm_apmCustomStackSize = 4096;

// 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 LowLevelIRQHandler (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" ));
    UnsubscribeTCU();
  }
  return TRUE;
}

void UnsubscribeTCU ( void )
{
  // Stops event generation, and gets remaining time
  u32 RemainingTimer
  adl_tcuStop ( TCUHandle, &RemainingTimer );
  // Un-subscribes from the TCU service
  adl_tcuUnsubscribe ( TCUHandle );
}

void adl_main (adl_InitType_e adlInitType)
{
  // Subscribes to the IRQ service
  IrqHandle = adl_irqSubscribe ( LowLevelIRQHandler, 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 );
  adl_tcuStart ( TCUHandle );
}
```

Event Capture Service Without Event Generation:

```
#include "adl_global.h"
#include “adl_tcu.h”
const u16 wm_apmCustomStackSize = 4096;

// Global variables
//Cyclic Timer pointer
adl_tmr_t *timer_ptr;
//Cyclic timer period
u16 timeout_period = 5;
// TCU service handle
s32 TCUHandle;
// Event counter to be provided to the API
u32 EventCounter;
// 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, &EventCounter };

void Timer_Handler (u8 Id, void* Context)
{
  // Periodically monitor the events counter
  TRACE (( 1, "Current events count: %d", EventCounter ));
}

void adl_main (adl_InitType_e adlInitType)
{
  // Subscribes to the TCU service, in Accurate Timer mode
  TCUHandle = adl_tcuSubscribe ( ADL_TCU_EVENT_CAPTURE, 0, 0, &Config, NULL );
  // Reset counter to 0, and starts event generation
  EventCounter = 0;
  // A cyclic timer is subscribed.
  timer_ptr = (adl_tmr_t*) adl_tmrSubscribe (TRUE, timeout_period,
            ADL_TMR_TYPE_100MS, (adl_tmrHandler_t) Timer_Handler);
  adl_tcuStart ( TCUHandle );
}
```

Event Capture Service With Event Generation:

```
#include "adl_global.h"
#include "adl_tcu.h"
const u16 wm_apmCustomStackSize = 4096;

// 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 LowLevelIRQHandler (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;
}

void UnsubscribeTCU ( void )
{
  // Stops event generation
  adl_tcuStop ( TCUHandle, NULL );
  // Un-subscribes from the TCU service
  adl_tcuUnsubscribe ( TCUHandle );
}

void adl_main (adl_InitType_e adlInitType)
{
  // Subscribes to the IRQ service
  IrqHandle = adl_irqSubscribe ( LowLevelIRQHandler, 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 );
}
```

Event Detection Service:

```
#include "adl_global.h"
#include "adl_tcu.h"

const u16 wm_apmCustomStackSize = 4096;

// 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 LowLevelIRQHandler (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;
}

void UnsubscribeTCU ( void )
```

```
{
  // Stops event generation
  adl_tcuStop ( TCUHandle, NULL );
  // Un-subscribes from the TCU service
  adl_tcuUnsubscribe ( TCUHandle );
}

void adl_main (adl_InitType_e adlInitType)
{
  // Subscribes to the IRQ service
  IrqHandle = adl_irqSubscribe ( LowLevelIRQHandler, 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 );
}
```

Summary

The following points have been covered in this chapter:

- **TCU service is used to serve the following needs of the real-time system.**
    - **Provides a way to define accurate timers with a precision in order of µs.**
    - **Allows to measure the number of edges(Events) occurred on a PIN in a given stipulated time.**
    - **Allows to detect a edge in a given time.**
- **There are 3 type of TCU services:**
    - **Accurate Timer Service**
    - **Event Detection Service**
    - **Event Capture Service**
- **The API provided by TCU service are:**
    - **adl_tcuSubscribe API is used to subscribe for the TCU service**
    - **adl_tcuUnsubscribe API is used to unsubscribe from the TCU service**
    - **adl_tcuStart API is used to start the TCU service**
    - **adl_tcuStop API is used to stop the TCU service**

CHAPTER 31

Power Consumption Modes

1. Objective

This chapter will introduce you to the different power consumption modes supported by embedded module, the steps to configure the low power modes and power consumed in low power modes

2. Introduction

The power consumption of a device is dependent on the interactions with the network and its peripherals. There are multiple ways to switch the module to a low power mode in order to reduce the power consumption.

The key contributions to the power consumption arise from the following:

- GSM or GPRS active connection with network (e.g. call, SMS)
- GSM/GPRS stack is active with no connection
- SIM is active
- Serial port access
- Embedded application execution

The embedded module consumes maximum power in connected mode i.e. when a connection is active with the network. When there is no active connection, the power consumption is lesser when compared to the connected mode.

3. Types of low power modes

The fact that not all of the features will be required all the time is taken as an advantage and low power modes are devised. These modes allow switching the embedded module to a state which disallows some of the features. The different modes are:

- Active mode with GSM stack ON
- Active mode with GSM stack OFF
- Sleep mode with GSM stack ON
- Sleep mode with GSM stack OFF
- Alarm Mode and
- OFF mode

3.1. Active mode with GSM stack ON

This is the default mode for embedded module. There is no feature restriction in this mode.

3.2. Active mode with GSM stack OFF

In this mode, SIM device and GSM/GPRS features like GSM voice or data call, SMS, GPRS data transfer are not available. The embedded application is running and the serial port remains active (AT commands are available). If any data has to be transmitted over the network, this mode has to be turned off.
To switch the embedded module to this mode use "AT+WBHV=1,1" command. The embedded module must be restarted in order to take the new behavior into account. To disable this mode, use the "AT+WBHV=1,0" command and restart the embedded module.

NOTE :
Active mode with GSM stack turned off can also be entered once using AT+CPOF and ON_OFF set to HIGH.

3.3. Sleep mode with GSM stack ON

Sleep mode is a low power mode in which the embedded module will have restricted access to the peripheral interfaces; thereby UARTs, USB, SPIs, I2C, GPIOs, ADCs and Buzzer are not available. This mode can be either enabled or disabled using the AT+W32K command. When this mode is activated, the embedded module requires 1 to 15 seconds to power down consumption. In this state, a 32 kHz internal clock is used during the inactivity stage. In this case, the embedded module will automatically wake up on unsolicited events such as GSM paging, external interruption, key press, alarm and Open AT timer expiration*. During the wake up period, the embedded module will have the same characteristic as the ACTIVE mode with GSM stack in idle in terms of power consumption and Open AT power processing. The embedded application will still be running.
The activation and deactivation of this mode is initiated by the customer device connected on the serial interface (DTE). The sleep mode can be activated with GSM stack not only by pulling UP the DTR pin but also on events such as external timers, soft strict timers and paging. To deactivate this mode DTR pin must be pulled DOWN.

3.4. Sleep mode with GSM stack OFF

In this mode, SIM device and GSM/GPRS features like GSM voice or data call, SMS, GPRS data transfer are not available. In addition, the embedded module has restricted access to peripheral interfaces; thereby UARTs, USB, SPIs, I2C, GPIOs, ADCs and Buzzer are not available.

The embedded module can be switched to this mode by using “AT+WBHV=1,2” command. The embedded module must be restarted in order to take the new behaviour into account. To disable this mode, use the AT command AT+WBHV=1,0.

In this mode, the embedded module can automatically wake up on unsolicited events such as external interruption, key press, alarm and Open AT timer expiration*.

3.5. Alarm mode

The only feature which is available in this mode is the alarm wake up. To use this mode, an alarm wake up has to be previously recorded by AT+CALA command before switching to this mode.

To activate the embedded module in this mode; use AT+CPOF command, when the ON/OFF switch is LOW. This mode can be deactivated by placing the ON/OFF switch to HIGH or after alarm wake up.

3.6. OFF mode

This mode is similar to Alarm mode. While using this command, the ON/OFF switch should be LOW. None of the functionality will be available in this mode. The embedded application will not be running.
This mode can be activated using the "AT+CPOF" command, when the ON/OFF switch is LOW. This mode can be deactivated by placing the ON/OFF switch to HIGH.

NOTE :
** Indicates that the Open AT timer expiration is nothing but the software strict timer expiration. The timer is strict if the parameter "Strict" of the API adl_tmrSubscribeExt is set to TRUE, which will awake the embedded module from the SLEEP mode with GSM stack ON when it occurs.*

4. Power Measurement

This section describes the various steps to be followed to make the power measurement. Figure below describes the hardware configuration.

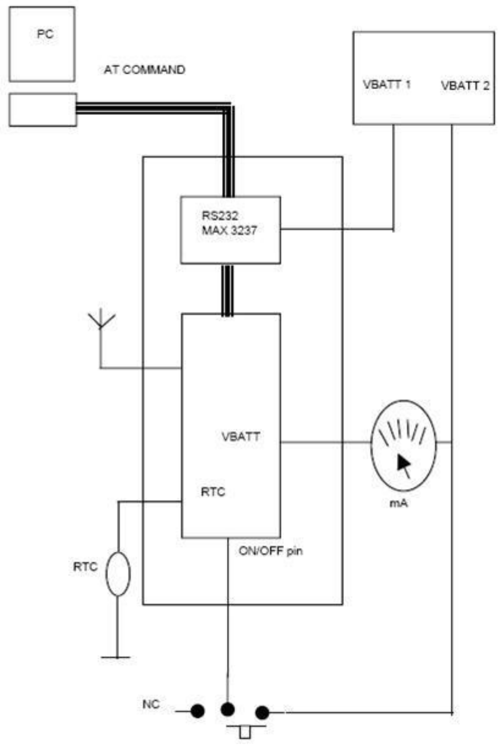
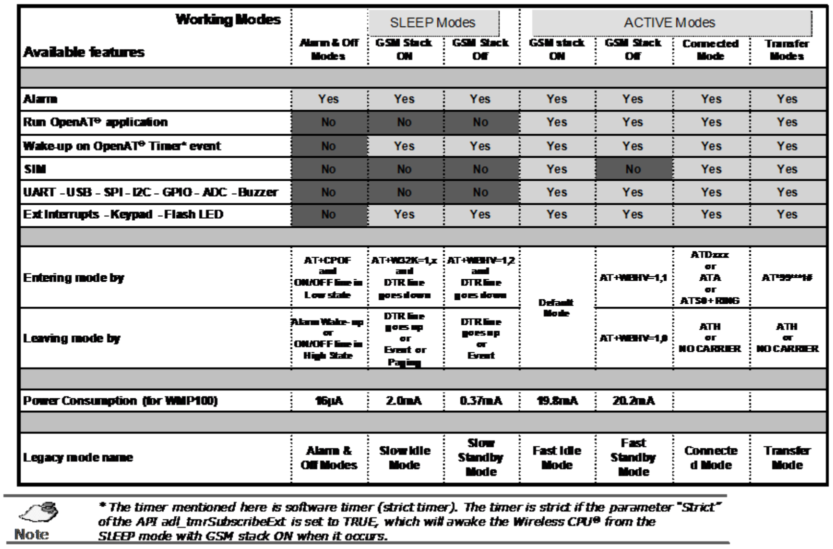


Figure 84: Hardware configuration of measuring power

In the figure above, VBATT 1 is used to supply only the MAX 3237 in order to avoid over consumption on the multimeter display. The consumption of this component is roughly 15 mA alone. VBATT 2 supplies power to the module in order to read the real value on the multimeter display. RTC is supplied a battery backup of 2.8 V. After configuring in the above mentioned manner, the value of the multimeter should be read to note the power consumption. Each of the commands should be given and the difference in power consumption can be noted.

5. Comparison of Low Power Modes

The table below gives a summary of the various power consumption modes and the different features available in these power consumption modes:



Working Modes / Available features		SLEEP Modes		ACTIVE Modes			
	Alarm & Off Modes	GSM Stack ON	GSM Stack Off	GSM stack ON	GSM Stack Off	Connected Mode	Transfer Modes
Alarm	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Run OpenAT® application	No	No	No	Yes	Yes	Yes	Yes
Wake-up on OpenAT® Timer* event	No	Yes	Yes	Yes	Yes	Yes	Yes
SIM	No	No	No	Yes	No	Yes	Yes
UART - USB - SPI - I2C - GPIO - ADC - Buzzer	No	No	No	Yes	Yes	Yes	Yes
Ext Interrupts - Keypad - Flash LED	No	Yes	Yes	Yes	Yes	Yes	Yes
Entering mode by	AT+CPOF and ON/OFF line in Low state	AT+W32K=1,x and DTR line goes down	AT+WBHV=1,2 and DTR line goes down	Default Mode	AT+WBHV=1,1	ATDxxx or ATA or ATS0+RING	AT*99***1#
Leaving mode by	Alarm Wake-up or ON/OFF line in High State	DTR line goes up or Event or Paging	DTR line goes up or Event		AT+WBHV=1,0	ATH or NO CARRIER	ATH or NO CARRIER
Power Consumption (for WMP100)	16µA	2.0mA	0.37mA	19.8mA	20.2mA		
Legacy mode name	Alarm & Off Modes	Slow Idle Mode	Slow Standby Mode	Fast Idle Mode	Fast Standby Mode	Connected Mode	Transfer Mode

Note: *The timer mentioned here is software timer (strict timer). The timer is strict if the parameter "Strict" of the API adl_tmrSubscribeExt is set to TRUE, which will awake the Wireless CPU® from the SLEEP mode with GSM stack ON when it occurs.*

Figure 85: Comparison of low power modes

Summary

The following points have been covered in this chapter:

- **Power consumption of a device is dependent on the interactions with the network and its peripherals. Open AT provides multiple ways to switch the module to a low power mode in order to reduce the power consumption**
- **The various modes supported are:**
  - **Active mode with GSM stack ON**
  - **Active mode with GSM stack OFF**
  - **Sleep mode with GSM stack ON**
  - **Sleep mode with GSM stack OFF**
  - **Alarm Mode and**
  - **OFF mode**

CHAPTER 32

Audio Service

1. Objective

This chapter will introduce you to the Audio service provided by the Open AT OS and how to use them in your Open AT Application.

2. Introduction

Embedded module is capable of managing following audio format:

- Single/Dual tone
- DTMF
- Melody
- PCM Audio Stream
- AMR/AMR-WB Audio Stream
- These audio formats could be used on following resources:
- Output
- Speaker
- Buzzer
- Outgoing voice call
- Input
- Microphone
- Incoming voice call

2.1. Single/Dual Tone

Single/Dual tone can be used to generate:

- Custom tones such as key press tone
- PSTN tones such as dial tone, busy tone

Single tone can be played on speaker or buzzer. However, the dual tone can only be played on speaker. Following parameters can be configured for these tones:

- Frequency in Hz
- 2 frequencies in case of dual tone
- Gain in dB
- Duration in multiple of 20 ms

2.2. DTMF

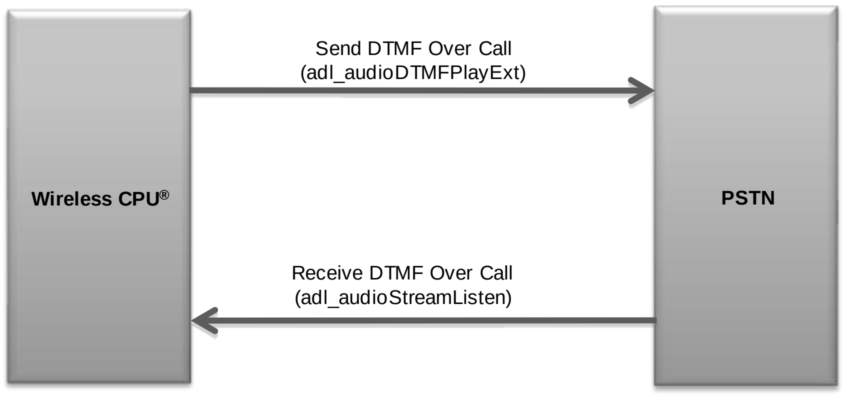
DTMF is a common method to send signaling information during a call. . This tone is a combination of two tones with pre-defined frequency. DTMF allows to send tones for 0-9,A-D,*,# characters. Open AT allows developer:

- To play the DTMF tones on speaker or voice call
- To process DTMF tones received through microphone or voice call

Following parameters can be configured for DTMF tones:

- Gain
- Duration

When DTMF tone is received, low level or high level interrupt handlers will be called as per the configuration of IRQ service.



Note: As per the configuration of IRQ service, low level or high level interrupts handlers will be called when DTMF is received

Figure 86 - Example for DTMF Tone

2.3. Melody

Melody is used to alert the user when:

- An incoming SMS is arrived
- An incoming call indication is received

Open AT allows defining a mono-tone melody and playing the same on speaker or buzzer. Following parameters can be configured for melody:

- Tempo of the melody (Speed)
- Number of times the melody can be played
- Gain

2.4. Audio Stream

PCM and AMR/AMR-WB are used to store digital representation of analog voice. Open AT OS provides Audio Service to capture this PCM/AMR/AMR-WB buffer through an interrupt.

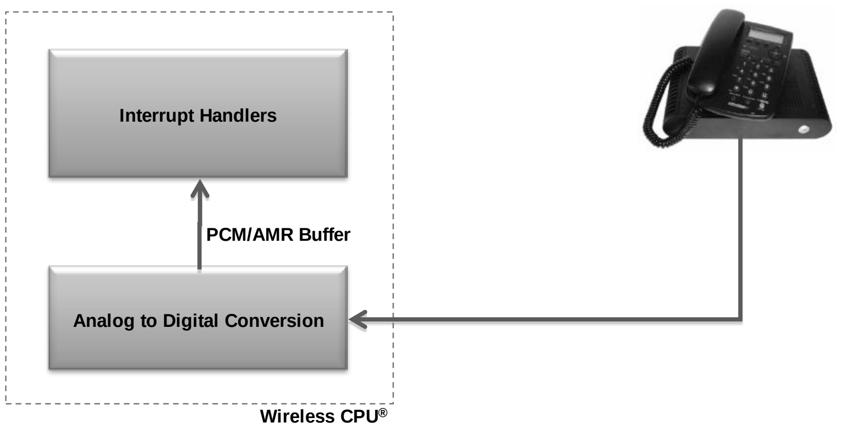


Figure 87 - Example for PCM/AMR/AMR-WB Audio Stream

Following parameters can be configured for Audio stream:

- Format of the stream
- PCM 16 bits/8 KHz
- PCM 16 bits/16 KHz
- AMR/AMR-WB

3. Header file

Open AT OS provides interfaces to handle audio resources and play or listen supported audio formats on these resources (single/dual tones, DTMF tones, melodies, audio streams, decoded DTMF streams).
Open AT Application should include adl_audio.h header file to use this service.

3.1. Subscribe to Audio Service

This API is used to subscribe to Audio service. This function allows subscribing to one of the available resources and specifying its behavior when another client attempts to subscribe it.

Prototype:

s32 adl_audioSubscribe (adl_audioResources_e audioResource, adl_audioEventHandler_f audioEventHandler, adl_audioResourceOption_e Options)

Parameters:

audioResource: This parameter defines the type of audio resource of the embedded module. 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). This parameter can have one of the following value:

```
typedef enum
{
  ADL_AUDIO_SPEAKER, // Speaker (output resource)
  ADL_AUDIO_BUZZER,  // Buzzer (output resource)
  ADL_AUDIO_MICROPHONE, // Current microphone (input resource)
  ADL_AUDIO_VOICE_CALL_RX, // Running voice call incoming channel
                           //(input resource)
  ADL_AUDIO_VOICE_CALL_TX  // Running voice call outgoing channel
                           //(output resource)
} adl_audioResources_e;
```

audioEventHandler: This is the call back function which will be called when events related to audio resource occurs. This is defined by the following format:

```
typedef void(*) adl_audioEventHandler_f(s32 audioHandle, adl_audioEvents_e Event)
```

Parameters:

audoHandle: This is the handle of the audio resource which is associated to the event.
Event: This parameter is the type of the event received. This parameter can have one of the following value:

```
typedef enum
{
  ADL_AUDIO_EVENT_NORMAL_STOP,
  ADL_AUDIO_EVENT_RESOURCE_RELEASED
} adl_audioEvents_e;
```

Parameters:

ADL_AUDIO_EVENT_NORMAL_STOP: This event is received when the pre-defined audio format play ends.
ADL_AUDIO_EVENT_RESOURCE_RELEASED: This event is received when the resource is automatically unsubscribed (ADL_AUDIO_RESOURCE_OPTION_ALLOW_PREEMPTION).
Options: This defines the audio resource subscription option. This parameter can have one of the following values:

```
typedef enum
{
  ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION = 0x00,
  ADL_AUDIO_RESOURCE_OPTION_ALLOW_PREEMPTION = 0x01
} adl_audioResourceOption_e;
```

Parameters:

ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION: When this option is used, the resource subscriber owns the resource until un-subscription time.
ADL_AUDIO_RESOURCE_OPTION_ALLOW_PREEMPTION: When this option is used, the resource can be used by other services such as incoming voice call melody, outgoing voice call tone play, SIM Toolkit application tone play. In this scenario, un-subscription happens automatically and the event ADL_AUDIO_EVENT_RESOURCE_RELEASED will be received in the event handler.

Returned Values:

Positive or null is returned if allocation succeeds.
ADL_RET_ERR_PARAM is returned if the parameter has an incorrect value.
ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the resource is already subscribed.
ADL_RET_ERR_NOT_SUPPORTED is returned if the resource is not supported.
ADL_RET_ERR_SERVICE_LOCKED is returned if this function is called from a low level interrupt handler.

3.2. Unsubscribe from Audio Service

This API is used to unsubscribe from an audio service. Note that it is not possible to unsubscribe from an audio resource if it is in running state. Hence, it is necessary to stop it using adl_audioStop API before unsubscribing the running resource.

Prototype:

```
s32 adl_audioUnsubscribe (s32 audioHandle )
```

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_NOT_SUBSCRIBED is returned if no audio resource has been subscribed.
ADL_RET_ERR_BAD_STATE is returned if an audio stream is listening or audio predefined signal is playing.
ADL_RET_ERR_SERVICE_LOCKED is returned if called from a low level interrupt handler.

3.3. Playing Single/Dual Tone

This API is used to play 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 when Open AT Application called adl_audioStop API.

Prototype:

s32 adl_audioTonePlay (s32 audioHandle, u16 Frequency1, s16 Gain1, u16 Frequency2, s16 Gain2, u32 Duration)

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.
Frequency1: This parameter sets the frequency for the 1st tone in Hz.
Gain1: This parameter sets the tone gain which will be applied to the 1st frequency value in dB.
Frequency2: This parameter sets the frequency for the 2nd tone in Hz.
Gain2: This parameter sets the tone gain which will be applied to the 2nd frequency value in dB.
Duration: This parameter sets the tone duration in ms. The value has to be a 20-ms multiple.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if parameters have an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if an audio stream is listening or audio predefined signal is playing on the required audio resource.
ADL_RET_ERR_BAD_HDL is returned if the audio resource is not allowed for tone playing.
ADL_RET_ERR_NOT_SUPPORTED is returned if the audio resource is not available for tone playing.
ADL_RET_ERR_SERVICE_LOCKED is returned if this function is called from a low level interrupt handler.

3.4. Playing Single/Dual Tone with a High Precision Gain

This API's functionality is same as that of adl_audioTonePlay API except that the gain values that are supported for this API is at a precision of 1/100dB. However for the adl_audioTonePlay API supports the gain in terms of 1dB.

Prototype:

s32 adl_audioTonePlayExt (s32 audioHandle, u16 Frequency1, s16 Gain1, u16 Frequency2, s16 Gain2, u32 Duration)

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.
Frequency1: This parameter sets the frequency for the 1st tone in Hz.
Gain1: This parameter sets the tone gain which will be applied to the 1st frequency value in 1/100 of dB.
Frequency2: This parameter sets the frequency for the 2nd tone in Hz.
Gain2: This parameter sets the tone gain which will be applied to the 2nd frequency value in 1/100 of dB.
Duration: This parameter sets the tone duration in ms. The value has to be a 20-ms multiple.

Returned Values:

OK is returned on success.

ADL_RET_ERR_PARAM is returned if parameters have an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if an audio stream is listening or audio predefined signal is playing on the required audio resource.
ADL_RET_ERR_BAD_HDL is returned if the audio resource is not allowed for tone playing.
ADL_RET_ERR_NOT_SUPPORTED is returned if the audio resource is not available for tone playing.
ADL_RET_ERR_SERVICE_LOCKED is returned if this function is called from a low level interrupt handler.

3.5. Playing DTMF Tone

This API is used to a DTMF tone on current speaker or on voice call TX (in communication only). Note that it is not possible to play DTMF tone on the buzzer. The specified output stops to play at the end of tone duration or when Open AT Application called adl_audioStop() API.

Prototype:

```
s32 adl_audioDTMFPlay (s32 audioHandle, ascii DTMF, s8 Gain, u32 Duration )
```

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.
DTMF: This parameter sets the DTMF to play (0-9,A-D,*,#).
Gain: This parameter sets the DTMF gain in dB.
Duration: This parameter sets the DTMF duration in ms. The value has to be a 20-ms multiple.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if parameters have an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if an audio stream is listening or audio predefined signal is playing on the required audio resource.
ADL_RET_ERR_BAD_HDL is returned if the audio resource is not allowed for DTMF playing.
ADL_RET_ERR_NOT_SUPPORTED is returned if the audio resource is not available for DTMF playing.
ADL_RET_ERR_SERVICE_LOCKED is returned if this function is called from a low level interrupt handler.

NOTE :
A DTMF can't be stopped on client request when DTMF is played on voice call TX.

3.6. Playing DTMF Tone with a High Precision Gain

This API's functionality is same as that of adl_audioDTMFPlay API except that the gain values that are supported for this API is at a precision of 1/100dB. However for the adl_audioDTMFPlay API supports the gain in terms of 1dB.

Prototype:

```
s32 adl_audioDTMFPlayExt (s32 audioHandle, ascii DTMF, s16 Gain, u32 Duration )
```

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.
DTMF: This parameter sets the DTMF to play (0-9,A-D,*,#).
Gain: This parameter sets the DTMF gain in 1/100 of dB.
Duration: This parameter sets the DTMF duration in ms. The value has to be a 20-ms multiple.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if parameters have an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if an audio stream is listening or audio predefined signal is playing on the required audio resource.
ADL_RET_ERR_BAD_HDL is returned if the audio resource is not allowed for DTMF playing.
ADL_RET_ERR_NOT_SUPPORTED is returned if the audio resource is not available for DTMF playing.
ADL_RET_ERR_SERVICE_LOCKED is returned if this function is called from a low level interrupt handler.

NOTE :
A DTMF can't be stopped on client request when DTMF is played on voice call TX.

3.7. Playing Melody

This API is used to play a defined melody on current speaker or buzzer. The specified output stops to play when the melody has been played the same number of time than that is specified in CycleNumber parameter or when Open AT Application called adl_audioStop API.

Prototype:

```
s32 adl_audioMelodyPlay (s32 audioHandle, u16 * MelodySeq, u8 Tempo, u8 CycleNumber,s8 Gain
)
```

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.
MelodySeq: This parameter sets the melody to be played. A melody is defined by an u16 table, where each element defines a note event, duration and sound definition. Note that the melody sequence has to end by a NULL value. This parameter is defined by the macro ADL_AUDIO_NOTE_DEF as mentioned below:

```
#define ADL_AUDIO_NOTE_DEF (ID, Scale, Duration )(((ID)+(Scale*12))<<8)+(Duration));
```

Parameters:

ID: This parameter corresponds to the note identification as mentioned below:

```
#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 DL_AUDIO_NO_SOUND 0xFF //No sound
```

Scale: This parameter defines the note scale (0 - 7).
Duration: This parameter defines the note duration. This parameter can have one of the following value:

```
#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_HEIGHT 0x02 //Height 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_HEIGHT 0x03 //Dotted height
```

Tempo: This parameter is the tempo in bpm (1 beat = 1 quarter note).
CycleNumber: This parameter defines the number of times the melody should be played. If it is not specified, the cycle number is infinite and melody should be stopped by client.
Gain: This parameter sets the melody gain in dB.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if parameters have an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if an audio stream is listening or audio predefined signal is playing on the required audio resource.
ADL_RET_ERR_BAD_HDL is returned if the audio resource is not allowed for melody playing.
ADL_RET_ERR_NOT_SUPPORTED is returned if the audio resource is not available for melody playing.
ADL_RET_ERR_SERVICE_LOCKED is returned if this function is called from a low level interrupt handler.

3.8. Playing Melody with a High Precision Gain

This API's functionality is same as that of adl_audioMelodyPlay API except that the gain values that are supported for this API is at a precision of 1/100dB. However for the adl_ audioMelodyPlay API supports the gain in terms of 1dB.

Prototype:

```
s32 adl_audioMelodyPlayExt (s32 audioHandle, u16 * MelodySeq, u8 Tempo, u8 CycleNumber, s16 Gain )
```

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.
MelodySeq: This parameter sets the melody to be played. A melody is defined by an u16 table, where each element defines a note event, duration and sound definition. Note that the melody sequence has to end by a NULL value. This parameter is defined by the macro ADL_AUDIO_NOTE_DEF as mentioned below:

```
#define ADL_AUDIO_NOTE_DEF (ID, Scale, Duration )(((ID)+(Scale*12))<<8)+(Duration));
```

Parameters:

ID: This parameter corresponds to the note identification as mentioned below:

```
#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 DL_AUDIO_NO_SOUND 0xFF //No sound
```

Scale: This parameter defines the note scale (0 - 7).
Duration: This parameter defines the note duration. This parameter can have one of the following value:

```
#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_HEIGHT 0x02 //Height 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_HEIGHT 0x03 //Dotted height
```

Tempo: This parameter is the tempo in bpm (1 beat = 1 quarter note).
CycleNumber: This parameter defines the number of times the melody should be played. If it is not specified, the cycle number is infinite and melody should be stopped by client.

Gain: This parameter sets the melody gain in frequency value in 1/100 of dB.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if parameters have an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if an audio stream is listening or audio predefined signal is playing on the required audio resource.
ADL_RET_ERR_BAD_HDL is returned if the audio resource is not allowed for melody playing.
ADL_RET_ERR_NOT_SUPPORTED is returned if the audio resource is not available for melody playing.
ADL_RET_ERR_SERVICE_LOCKED is returned if this function is called from a low level interrupt handler.

NOTE :
1. Memory space has to be allocated for the audio stream buffer before playing starts and it has to be released after playing stops.
2. Only audio PCM/AMR/AMR-WB sample can be played.

3.9. Playing Audio Stream

This function is used to play an audio sample stream on the current speaker or on voice call TX.

Prototype:

```
s32 adl_audioStreamPlay (s32 audioHandle, adl_audioFormats_e audioFormat, s32 LowLevelIRQHandle, s32 HighLevelIRQHandle, void * buffer )
```

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.
audioFormat: This parameter defines the audio stream formats for audio stream playing/listening as mentioned below:

```
typedef enum _adl_audioFormats_e
{
   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
```

LowLevelIrqHandle: This parameter is the IRQ handle returned by IRQ service for low level interrupts.
HighLevelIrqHandle: This parameter is the IRQ handle returned by IRQ service for high level interrupts.
buffer: This parameter is a buffer pointer which has audio stream data to be played. The below mentioned 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.

```
typedef struct
{
  adl_audioFormats_e audioFormat
  adl_audioStreamDataBuffer_u * DataBuffer
  bool * BufferReady
  bool * BufferEmpty
} adl_audiostream_t;
```

Parameters:

audioFormat: This parameter defines the audio stream formats for audio stream playing/listening as mentioned below:

```
typedef enum _adl_audioFormats_e
{
    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
```

DataBuffer: This parameter defines different types of buffers, which are used according to the audio format when an audio stream interrupt occurs.

- This field stores audio sample during an audio stream listening or decoded DTMF during an audio DTMF stream listening.
- It contains audio sample to play during an audio stream playing. This is defined by the adl_audioStreamDataBuffer_u structure as mentioned below:

```
typedef union
{
u8 PCMData [1]
u8 AMRData [1];
adl_audioDecodedDtmf_u DTMFData
} adl_audiostreamDataBuffer_u;
```

Parameters:

PCMData: This parameter is the PCM stream data buffer. This buffer is used when playing or listening to an audio PCM stream.
AMRData: This contains the AMR data as received on audio stream.

DTMFData: This parameter is the DTMF stream data buffer. This buffer stores decoded DTMF when listening to an audio DTMF stream according to the decoding mode which is used
BufferReady: When an audio stream is played, each time an interrupt 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. This pointer is initialized only when an audio stream is played. Currently, it is used only for PCM stream playing.
BufferEmpty: When an audio stream is played, this flag is set to TRUE when empty data buffer is played (for example, when an interrupt 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.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if parameters have an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if an audio stream is listening or audio predefined signal is playing on the required audio resource.
ADL_RET_ERR_BAD_HDL is returned if the audio resource is not allowed for audio stream playing or if interrupt handler identifiers are invalid.
ADL_RET_ERR_NOT_SUPPORTED is returned if the audio resource is not available for audio stream playing.
ADL_RET_ERR_SERVICE_LOCKED is returned if called from a low level interrupt handler.

NOTE :
1. It is mandatory to specify low level IRQ handler to make this API work properly. Hence IRQ service should be subscribed to the low level interrupts with ADL_IRQ_OPTION_AUTO_READ option.
2. High level IRQ handler is a optional parameter.
3. 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.
4. 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).
5. For audio interrupt subscription ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT option is not available.
NOTE : Memory space has to be allocated for the audio stream buffer before playing starts and it has to be released after playing stops.

3.10. Listening to DTMF Tone/Audio Stream

This API is used to listen to a DTMF tone or an audio sample from microphone or voice call RX.

Prototype:

s32 adl_audioStreamListen (s32 audioHandle, adl_audioFormats_e audioFormat, s32 LowLevelIRQHandle, s32 HighLevelIRQHandle, void * buffer)

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.
audioFormat: This parameter defines the audio stream formats for audio stream playing/listening as mentioned below:

```
typedef enum _adl_audioFormats_e
{
   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;
```

LowLevelIrqHandle: This parameter is the IRQ handle returned by IRQ service for low level interrupts.
HighLevelIrqHandle: This parameter is the IRQ handle returned by IRQ service for high level interrupts.
buffer: This parameter is a buffer pointer which has audio stream data to be played. The below mentioned 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.

```
typedef struct
{
  adl_audioFormats_e audioFormat
  adl_audioStreamDataBuffer_u * DataBuffer
  bool * BufferReady
  bool * BufferEmpty
} adl_audiostream_t;
```

Parameters:

audioFormat: This parameter defines the audio stream formats for audio stream playing/listening as mentioned below:

```
typedef enum _adl_audioFormats_e
{
   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;
```

DataBuffer: This parameter defines different types of buffers, which are used according to the audio format when an audio stream interrupt occurs.

This field stores audio sample during an audio stream listening or decoded DTMF during an audio DTMF stream listening.
It contains audio sample to play during an audio stream playing. This is defined by the adl_audioStreamDataBuffer_u structure as mentioned below:

```
typedef union
{
u8 PCMData [1];
u8 AMRData [1];
adl_audioDecodedDtmf_u DTMFData;
} adl_audiostreamDataBuffer_u;
```

Parameters:

PCMData: This parameter is the PCM stream data buffer. This buffer is used when playing or listening to an audio PCM stream.
AMRData: This contains the AMR data as received on stream.
DTMFData: This parameter is the DTMF stream data buffer. This buffer stores decoded DTMF when listening to an audio DTMF stream according to the decoding mode which is used
BufferReady: When an audio stream is played, each time an interrupt 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. This pointer is initialized only when an audio stream is played. Currently, it is used only for PCM stream playing.
BufferEmpty: When an audio stream is played, this flag is set to TRUE when empty data buffer is played (for example, when an interrupt 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.

Returned Values:

OK is returned on success.
ADL_RET_ERR_PARAM is returned if parameters have an incorrect value.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if an audio stream is listening or audio predefined signal is playing on the required audio resource.
ADL_RET_ERR_BAD_HDL is returned if the audio resource is not allowed for audio stream playing or if interrupt handler identifiers are invalid.
ADL_RET_ERR_NOT_SUPPORTED is returned if the audio resource is not available for audio stream playing.
ADL_RET_ERR_SERVICE_LOCKED is returned if called from a low level interrupt handler.

NOTE :
1. Low level IRQ handler is a optional parameter if high level IRQ handler is specified.
2. High level IRQ handler is a optional parameter if low level IRQ handler is specified.
3. 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.
4. 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).
5. For audio interrupt subscription, ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT option is not available..

3.11. Stopping to Play/Listen

This API is used 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 or an audio AMR 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.

Event ADL_AUDIO_EVENT_NORMAL_STOP will not be sent consequently when play/listen is stopped successfully using this API.

Prototype:

s32 adl_audioStop (s32 audioHandle)

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_BAD_STATE is returned if no audio process is running on the required audio resource.
ADL_RET_ERR_SERVICE_LOCKED is returned this function is if called from a low level interrupt handler.

3.12. Configuring Audio Options

This API is used for configuring audio option according to audio resource and option type specified.

Prototype:

s32 adl_audioSetOption (s32 audioHandle, adl_audioOptionTypes_e audioOption, s32 value)

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.
audioOption: This parameter specifies the configuration parameter. This parameter is defined as mentioned below:
ADL_AUDIO_DTMF_DETECT_BLANK_DURATION: This is a DTMF decoding option (u16); it allows defining 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 has to be a 10-ms multiple.

If a NULL value is specified, DTMF decoder will be in Raw mode (default), Raw data's coming from DTMF decoder are sent every 20 ms via interrupt handlers. This mode implies to implement an algorithm in order to detect the good DTMF. (Refer to adl_audioDecodedDtmf_u 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.
ADL_AUDIO_AMR_SPEECH_CODEC_RATE: Allows defining which codec rate will be used for AMR stream playing (adl_audioAmrCodecRate_e).
ADL_AUDIO_AMR_MIXED_VOICE: Allows defining if the AMR sample should be mixed to the voice when an AMR audio sample is played (bool).
ADL_AUDIO_AMR_BUFFER_SIZE: Allows defining the buffer type to allocate for playing or listening to on an AMR stream (u32).
value: This defines the value to be set.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_PARAM is returned if parameters have an incorrect value.

3.13. Retrieving Audio Options

This API is used for get the value of the different audio options according to audio resource and option type specified.

Prototype:

s32 adl_audioGetOption (s32 audioHandle, adl_audioOptionTypes_e audioOption, s32 * value)

Parameters:

audioHandle: This is the audio handle returned by the adl_audioSubscribe API.
audioOption: This parameter specifies the configuration parameter. This parameter can be set to one of the value as mentioned below:
ADL_AUDIO_DTMF_DETECT_BLANK_DURATION: This is a DTMF decoding option (u16); it allows defining 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 has to be a 10-ms multiple.
If a NULL value is specified, DTMF decoder will be in Raw mode (default), Raw data's coming from DTMF decoder are sent every 20 ms via interrupt handlers. This mode implies to implement an algorithm in order to detect the good DTMF. (Refer to adl_audioDecodedDtmf_u 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.
ADL_AUDIO_MAX_FREQUENCY: This parameter is the maximum frequency allowed to be played on the required output resource in Hz. The data type of the parameter is u16.
ADL_AUDIO_MIN_FREQUENCY: This parameter is the minimum frequency allowed to be played on the required output resource in Hz(u16).

ADL_AUDIO_MAX_GAIN: This is the maximum gain which can be set to play a pre-defined audio format using adl_audioDTMFPlayExt/adl_audioTonePlayExt/adl_audioMelodyPlayExt.The returned gain value is defined in 1/100 of dB (s16).
ADL_AUDIO_MIN_GAIN: This is the minimum gain which can be set to play a pre-defined audio format using adl_audioDTMFPlayExt/adl_audioTonePlayExt/adl_audioMelodyPlayExt.The returned gain value is defined in 1/100 of dB (s16).
ADL_AUDIO_MAX_DURATION: This is the maximum duration which can be set to play a DTMF tone or a single/dual tone using adl_audioDTMFPlay/adl_audioTonePlay. The returned duration value is defined in ms (u32).
ADL_AUDIO_MIN_DURATION: This is the minimum duration which can be set to play a DTMF tone or a single/dual tone using adl_audioDTMFPlay/adl_audioTonePlay. The returned duration value is defined in ms (u32).
ADL_AUDIO_MAX_NOTE_VALUE: This is the maximum duration for a note (tempo) which can be set to play play a melody using adl_audioMelodyPlay. This value is the maximal value which can be defined with ADL_AUDIO_NOTE_DEF macro (u32).
ADL_AUDIO_MIN_NOTE_VALUE: This is the minimum duration for a note (tempo) which can be set to play play a melody using adl_audioMelodyPlay. This value is the maximal value which can be defined with ADL_AUDIO_NOTE_DEF macro (u32).
ADL_AUDIO_DTMF_RAW_STREAM_BUFFER_SIZE: This is 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: This is 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: This is the user 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 (u8)
ADL_AUDIO_PCM_16K_16B_MONO_BUFFER_SIZE: This is the user 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: Allows knowing if AMR Wideband codec rates are available (bool).
ADL_AUDIO_AMR_SPEECH_CODEC_RATE: Allows getting codec rate that will be used for AMR stream playing (adl_audioAmrCodecRate_e).
ADL_AUDIO_AMR_MIXED_VOICE: Allows checking if the AMR sample should be mixed to the voice when an AMR audio sample is played (bool).
ADL_AUDIO_AMR_BUFFER_SIZE: Allows getting the buffer type to allocate for playing or listening to on an AMR stream (u32).
value: This defines the option value according to audio option which has been set.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the provided handle is unknown.
ADL_RET_ERR_PARAM is returned if parameters have an incorrect value.

4. Sample Code

4.1. Playing Tone

```
#include "adl_global.h"
#include "adl_audio.h"
const u16 wm_apmCustomStackSize = 4096;

// 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 );
}
```

4.2. Playing DTMF

```
#include "adl_global.h"
#include "adl_audio.h"

const u16 wm_apmCustomStackSize = 4096;

// 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);
}
```

4.3. Playing Melody

```
#include "adl_global.h"
#include "adl_audio.h"

const u16 wm_apmCustomStackSize = 4096;

// 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_HEIGHTH),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);
}
```

4.4. Playing Audio Stream

```
#include "adl_global.h"
#include "adl_audio.h"

const u16 wm_apmCustomStackSize = 4096;

// 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;
  adl_irqConfig_t Config;

  // 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 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
//Subscribe to low lever IRQ
MyIRQHandle = adl_irqSubscribeExt (MyLowLevelIRQHandler,ADL_IRQ_NOTIFY_LOW_LEVEL, &Config );
  // Play an audio PCM stream
  Ret = adl_audioStreamPlay( handle, ADL_AUDIO_PCM_MONO_8K_16B, MyIRQHandle, 0, StreamBuffer);
}
```

4.5. Listening Audio Stream

```
#include "adl_global.h"
#include "adl_audio.h"

const u16 wm_apmCustomStackSize = 4096;

// audio resource handle
s32 handle;
// audio stream buffer
void * StreamBuffer;
// Low level interrupt 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
 adl_irqConfig_t Config;
// 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 current microphone
 handle = adl_audioSubscribe ( ADL_AUDIO_MICROPHONE, MyAudioEventHandler ,
ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );
 // Memory allocation
 Ret = adl_audioGetOption (handle, ADL_AUDIO_PCM_8K_MONO_BUFFER_SIZE, &BufferSize )
 StreamBuffer = adl_memGet( BufferSize); // release memory after audio stream listening
 //Subscribe to low lever IRQ
 MyIRQHandle = adl_irqSubscribeExt (MyLowLevelIRQHandler,ADL_IRQ_NOTIFY_LOW_LEVEL,
                 &Config );

 // Listen to audio DTMF stream
 Ret = adl_audioStreamListen( handle, ADL_AUDIO_DTMF, MyIRQHandle, 0, StreamBuffer);
}
```

4.6. Stop Playing Tone

```
#include "adl_global.h"
#include "adl_audio.h"

const u16 wm_apmCustomStackSize = 4096;

// 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 );
}
```

Summary

The following points have been covered in this chapter:

- **Audio service is used 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 API provided by Audio service are:**
    - **adl_audioSubscribe API to subscribe to an audio resource**
    - **adl_audioUnsubscribe API to unsubscribe from an audio resource**
    - **adl_audioTonePlay API to play a single/dual tone**
    - **adl_audioDTMFPlay API to play a DTMF tone**
    - **adl_audioMelodyPlay API to play a melody**
    - **adl_audioTonePlayExt API to play a single/dual tone with high precision gain**
    - **adl_audioDTMFPlayExt API to play a DTMF tone with high precision gain**
    - **adl_audioMelodyPlayExt API to play a melody with high precision gain**
    - **adl_audioStreamPlay API to play an audio stream**
    - **adl_audioStreamListen API to listen to an audio stream**
    - **adl_audioStop API to stop playing or listening**
    - **adl_audioSetOption API to set audio options**
    - **adl_audioGetOption API to get audio options**

CHAPTER 33

VariSpeed

1. Objective

This chapter explains the APIs that can be used to change the processor speed.

2. Introduction

Open AT provides VariSpeed feature which can be used to boost the processor clock from 26 MHz (standard mode) to 104 MHz(boost mode) without even resetting the embedded module. However, the power consumption of the embedded module increases when operating at higher frequency such as 104 MHz.

The boost mode can be used when application needs to perform complex operation which will take more time. For e.g. a data related application uses following approach for management of data:

- Data Acquisition: Retrieves data from various sources
- Data Processing: Performs complicated algorithms on the input data and creates the output data.
- Data Transfer: Sends the output data to the server.

Here the Data Acquisition and Data Transfer can be done in normal mode. Data processing as it involves complex processing can be in boost mode to get more performance.

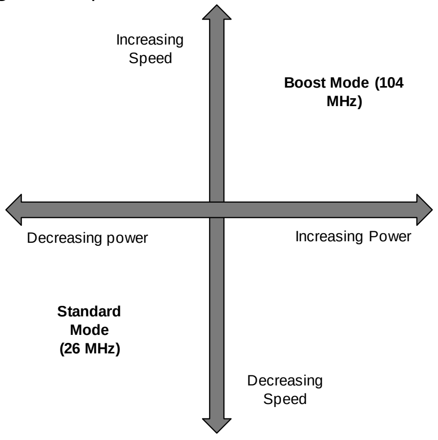


Figure 88 - Vari-Speed Feature

3. Header file

Open AT OS provides interfaces that can be used to boost the processor clock. Open AT Application should include adl_vs.h header file to use these interfaces.

3.1. Subscribe to VariSpeed Service

This API is used to subscribe to VariSpeed service.

Prototype:

s32 adl_vsSubscribe (void)

Parameters:

None

Returned Values:

A positive or null value is returned on success:
A negative error value is returned otherwise:
ADL_RET_ERR_ALREADY_SUBSCRIBED is returned if the service has already been subscribed.
ADL_RET_ERR_NOT_SUPPORTED is returned if the Real Time enhancement feature is not enabled on the embedded module.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

3.2. Unsubscribe from VariSpeed Service

This API is used to unsubscribe from VariSpeed service.

Prototype:

s32 adl_vsUnsubscribe (s32 VsHandle)

Parameters:

VsHandle: This parameter is the handle returned by adl_vsSubscribe API.

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the supplied handle is unknown.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

3.3. Change the Processor Speed

This API is used to change the processor clock speed. The clock speed can be:

26 MHz
104 MHz

Prototype:

```
s32 adl_vsSetClockMode (s32 VsHandle, adl_vsMode_e ClockMode )
```

Parameters:

VsHandle: This parameter is the handle returned by adl_vsSubscribe API.
ClockMode: This parameter defines the clock mode as defined in the enum adl_vsMode_e. This enum is defined below:

```
typedef enum _adl_vsMode_e
{
  ADL_VS_MODE_STANDARD,   // Standard clock mode(26 MHz)
  ADL_VS_MODE_BOOST,      // Boost clock mode(104 MHz)
  ADL_VS_MODE_LAST        // Reserved for internal use
} adl_vsMode_e;
```

Returned Values:

OK is returned on success.
ADL_RET_ERR_UNKNOWN_HDL is returned if the supplied handle is unknown.
ADL_RET_ERR_PARAM is returned if the supplied clock mode value is wrong.
ADL_RET_ERR_SERVICE_LOCKED is returned if the function is called from a low level interrupt handler.

4. Sample Code

```
#include "adl_global.h"
#include "adl_vs.h"

const u16 wm_apmCustomStackSize = 4096;

// Global variable: VariSpeed service handle
s32 MyVariSpeedHandle;
adl_tmr_t *timer_ptr;
u16 timeout_period = 5;
static counter=0;

void Timer_Handler (u8 Id, void* Context)
{
 TRACE ((1,"Inside timer handler. Timer id is %d", Id));
 counter++;
 adl_atSendResponse (ADL_AT_UNS,"\r\nCyclic Timer expired");
 if (counter==50)
 {
    adl_tmrUnSubscribe (timer_ptr, (adl_tmrHandler_t)
    Timer_Handler, ADL_TMR_TYPE_100MS);
    // Un-subscribe from the VariSpeed service
    adl_vsUnsubscribe ( MyVariSpeedHandle );
 }
}
void adl_main (adl_InitType_e adlInitType)
{
 // A cyclic timer is subscribed.
 timer_ptr = (adl_tmr_t*) adl_tmrSubscribe (TRUE, timeout_period,
          ADL_TMR_TYPE_100MS, (adl_tmrHandler_t) Timer_Handler);

 // Subscribe to the VariSpeed service
 MyVariSpeedHandle = adl_vsSubscribe();

 // Enter the boost mode
 adl_vsSetClockMode ( MyVariSpeedHandle, ADL_VS_MODE_BOOST )
}
```

Summary

The following points have been covered in this chapter

- **Open AT OS RTOS feature allows user to boost the processor clock from 26 MHz to 104 MHz without even resetting the embedded module.**
- **Following APIs are available for vari-speed service:**
    - **adl_vsSubscribe to subscribe to vari-speed service**
    - **adl_vsUnsubscribe to unsubscribe from vari-speed service**
    - **adl_vsSetClockMode to change the processor clock speed**

CHAPTER 34

Internet Plug-in for Open AT

1. Introduction

Welcome to the Internet Plug-in for Open AT. The following chapters will guide you through the features of the Internet Plug-in provided into the OASiS. Clear examples, diagrams, exercises and tips will help you easily understand and use the Internet Plug-In.

If you require additional information, please contact your Sierra Wireless representative.

2. Overview of TCP/IP

TCP/IP is a set of protocols that allows cooperating computers to share resources across a network. This defacto UNIX standard is the protocol used for the Internet, and is the global standard for communications. Every node in a TCP/IP network requires an IP address which is either permanently assigned or dynamically assigned at startup.
The TCP/IP protocol suite is implemented in the following 4 layers, similar in construct, but different in layer combination from the OSI 7 layer model:

- Physical layer
- Network Layer
- Transport Layer
- Application layer

The following image depicts the protocol layer architecture.

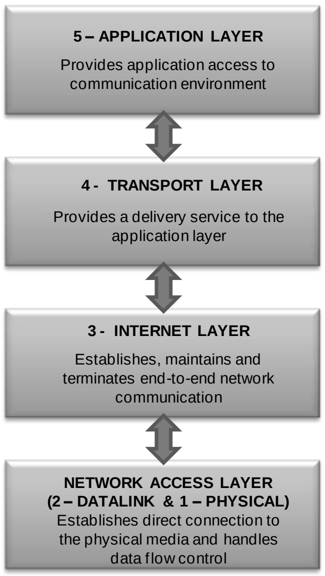

5 – APPLICATION LAYER
Provides application access to communication environment
4 - TRANSPORT LAYER
Provides a delivery service to the application layer
3 - INTERNET LAYER
Establishes, maintains and terminates end-to-end network communication
NETWORK ACCESS LAYER
(2 – DATALINK & 1 – PHYSICAL)
Establishes direct connection to the physical media and handles data flow control

Figure 89 - Architecture of TCP/IP

The message (data) to be transmitted after an application initiates a communication is passed through different layers, until it reaches the physical link to be used for data transmission. Each layer attaches its own header to the data packet. At the receiving end, the corresponding protocol layers unpack the data and pass it on to the higher layers.

TCP/IP is composed of two parts: TCP (Transmission Control Protocol) and IP (Internet Protocol). IP is the network layer protocol of the TCP/IP suite and is responsible for moving packets of data from node to node. IP forwards each packet based on a four-byte destination address (the IP number).

TCP is a connection-oriented and reliable transport layer protocol. This protocol is used for all the session-oriented applications. TCP is responsible for verifying the correct delivery of data between client and server. It tests for errors and requests retransmission if necessary.

UDP is an alternative to TCP; it is a connectionless protocol that can be used to send datagrams to the destination. Sierra Wireless's TCP/IP software includes UDP implementation too.

3. Objective

This chapter gives an introduction to the Internet Plug-In software which has been implemented on top of the Open AT OS package provided by Sierra Wireless. This chapter also explains various application protocols supported by the Internet Plug-in and its interfaces.

4. Why TCP/IP on Sierra Wireless embedded module

Sierra Wireless embedded module support GPRS connectivity and data that is received and sent over GPRS has to be in the form of IP packets. In order to understand the protocol and to receive and send data as IP packets, an

application can run on an external device connected to the Sierra Wireless embedded module. This application will decode the IP packets to understand the data received from the network, and also encode the data as IP packets and send them back to the network.
Sierra Wireless's TCP/IP software removes the need for any such external application. Sierra Wireless has provided the TCP/IP software along with OASiS using which, an Open AT Application can understand data coming over the GPRS network.

Thus, you can create intelligent internet-enabled applications directly on the embedded module. This will shorten your development time and reduce materials cost.

5. TCP/IP software

The TCP/IP software provides the following protocols:

- TCP Sockets
- UDP Sockets
- PING Service
- FTP Service
- HTTP Service
- SMTP Service
- POP3 Service
- Bearer management using PPP and GPRS.

Sierra Wireless provides an Internet Plug-in (Also known as WIP library) in which TCP/IP protocols have been implemented.

6. Architecture of Internet Plug-In

This is an Open AT Plug-In library which provides a set of APIs to implement TCP/IP-related functionality. These APIs can be used from inside an Open AT Application to connect using TCP/IP. The following figure depicts the architecture of TCP/IP stack.

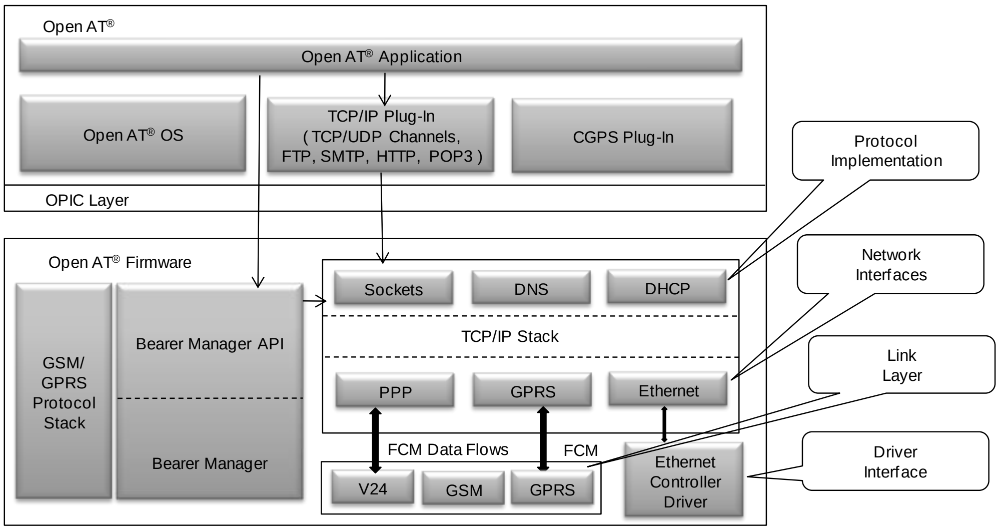


Figure 90 - Architecture of Internet Plug-In

The basic functionalities(TCP/UDP) will be implemented in Firmware. The FTP/SMTP/POP3/HTTP functionalities will be implemented in the Open AT Plug-In WIP library. Due to this, the applications which require only the basic TCP/UDP sockets, will be smaller in size.

An Firmware also has the implementation of,

- Bearer manager which is responsible for configuring and activating the different bearers such as UARTs, GSM, GPRS and Ethernet.
- DNS resolver responsible for resolving name to IP address resolution.
- DHCP client which is used by Ethernet interface when auto configuration is selected.
- The APIs available in Internet Plug-In are available in two flavors:
- APIs without any options
- APIs with advanced configurable options

7. Internet Plug-In APIs without options

These APIs uses default configuration set by Internet Plug-In. These APIs are easy to use and provides an fast access to Internet Plug-In. For e.g. wip_netInit() API can be used instead of wip_netInitOpts() to initialize the Internet Plug-In with default initialization parameters. These default parameters include number of sockets (by default 8) etc.

8. Internet Plug-In APIs with advanced configurable options

These APIs allows to configure the Internet Plug-In with advanced options. These APIs provides much more control over the Open AT Application. For e.g. wip_netInitOpts() can be used instead of wip_netInitOpts() to initialize the Internet Plug-In with advanced options such as number of sockets.

Summary

The following points have been covered in this chapter

- **Sierra Wireless provides a Internet plug-in which implements TCP/IP protocols**
- **Internet Plug-In is available as a plug-in**
- **Internet Plug-In APIs are available in two flavours**
  - **APIs with default options**
  - **APIs with configurable options**

CHAPTER 35

IP Stack Initialization

Internet Plug-In must be initialized before it can be used in an Open AT Application. During initialization, Internet Plug-In initializes the local data structures used inside in the Internet Plug-In and also allows to configure the following parameters:

- Number of sockets
- Number of buffers
- DNS parameters
- TCP/IP protocol options such as Time to live parameter

The header file that must be included to use initialization related APIs is “wip_net.h”.

1. Objective

This chapter describes the APIs that you can use to initialize the Internet Plug-In. This includes the APIs,

- To initialize the Internet Plug-In with default configuration options
- To initialize the Internet Plug-In with advanced configurable options
- To configure the protocol specific options
- To release the resource allocated during initialization

2. Initialization APIs

2.1. Initialization with default options

The wip_netInit() API can be used to initialize the Internet Plug-In with default options.

Prototype:

```
s8 wip_netInit( void)
```

Parameters:

None

Returned Values:

This function returns a value 0, if initialization was successful.

On error following values are returned:
WIP_NET_ERR_NO_MEM: An error was encountered during memory allocation for initialization related resources. This error code will be returned only if the application has subscribed for the ADL Error services. Otherwise, the embedded Module will reset.

2.2. Initialization with configurable advanced options

The wip_netInitOpts() API can be used to initialize the Internet Plug-In with user defined advanced option such as number of sockets.

Prototype:

s8 wip_netInitOpts(s32 opt, ...)

Parameters:

opt: This specifies the first option in the list of options.
…: List of options supported by this API. All options must be followed by a variable. In case of get operation, value will be copied to the variable while in case of set operation, value will be taken from the variable. The options supported by this API are:

WIP_NET_OPT_TCP_REXMT_MAX: Maximum time between TCP retransmissions. Default value for this parameter is 64 seconds.
WIP_NET_OPT_TCP_REXMT_MAXCNT: Maximum number of retransmissions. Default value for this parameter is 12
WIP_NET_OPT_IP_FORWARD: Activating IP forwarding. Default value for this parameter is FALSE
WIP_NET_OPT_IP_NAT_TO_TCP : TCP flow timeout. Default value for this parameter is 15 seconds.
WIP_NET_OPT_IP_NAT_TO_TCP_FIN: TCP FIN flow timeout. Default value for this parameter is 2 seconds.
WIP_NET_OPT_IP_NAT_TO_UDP: UDP flow timeout. Default value for this parameter is 5 seconds.
WIP_NET_OPT_IP_NAT_TO_ICMP: ICMP flow timeout. Default value for this parameter is 2 seconds.

The list of options should be terminated by a macro WIP_NET_OPT_END.

Returned Values:

This function returns a value 0, if initialization was successful.
On error following values are returned:
WIP_NET_ERR_OPTION: Incorrect configuration parameter entered
WIP_NET_ERR_PARAM: Incorrect configuration parameter value entered
WIP_NET_ERR_NO_MEM: An error was encountered during memory allocation for initialization related resources. Note that this value is returned only if Open AT Application is subscribed to Error service(adl_errSubscribe()) otherwise, the embedded module restarts.

2.3. Release the resources allocated during initialization

The wip_netExit() API can be used to release the resources allocated during initialization of the Internet Plug-In. Note that all the bearers which are opened should be closed before calling this function.

Prototype:

s8 wip_netExit(void)

Parameters:

None

Returned Values:

This function always returns a value 0

2.4. Configuration of TCP/IP protocol options

The wip_netSetOpts() API can be used to configure the TCP/IP protocol specific options such as time to live for IP packets.

Prototype:

s8 wip_netSetOpts(s32 opt, ...)

Parameters:

opt: This specifies the first option in the list of options.
...: List of options supported by this API. All options must be followed by a variable/constant.The option/value pair will define what value is assigned for a particular parameter. The options supported by this API are:

WIP_NET_OPT_IP_TTL: Time to live for outgoing datagram
WIP_NET_OPT_IP_TOS: Default type of service for IP datagram
WIP_NET_OPT_IP_FRAG_TIMEO: Time to live for incomplete fragments
WIP_NET_OPT_TCP_MAXINITWIN: Number of segments for initial TCP window
WIP_NET_OPT_TCP_MIN_MSS: Minimum segment size for off-link connection
WIP_NET_OPT_TCP_REXMT_MAX: Maximum time between TCP retransmissions. Default value for this parameter is 64 seconds.
WIP_NET_OPT_TCP_REXMT_MAXCNT: Maximum number of retransmissions. Default value for this parameter is 12
WIP_NET_OPT_IP_FORWARD: Activating IP forwarding. Default value for this parameter is FALSE
WIP_NET_OPT_IP_NAT_TO_TCP : TCP flow timeout. Default value for this parameter is 15 seconds.
WIP_NET_OPT_IP_NAT_TO_TCP_FIN: TCP FIN flow timeout. Default value for this parameter is 2 seconds.
WIP_NET_OPT_IP_NAT_TO_UDP: UDP flow timeout. Default value for this parameter is 5 seconds.
WIP_NET_OPT_IP_NAT_TO_ICMP: ICMP flow timeout. Default value for this parameter is 2 seconds.

The list of options should be terminated by a macro WIP_NET_OPT_END.

Returned Values:

This function returns a value 0, if successful.

On error following values are returned:
WIP_NET_ERR_OPTION: Incorrect configuration parameter entered
WIP_NET_ERR_PARAM: Incorrect configuration parameter value entered

2.5. Getting the current configuration of TCP/IP protocol options

The wip_netGetOpts() API can be used to get the current configuration done for the TCP/IP protocol specific options such as time to live for IP packets.

Prototype:

s8 wip_netGetOpts(s32 opt, ...)

Parameters:

opt: This specifies the first option in the list of options.
...: List of options supported by this API. All options must be followed by a variable. In case of get operation, value will be copied to the variable. The options supported by this API are same as mentioned for wip_netSetOpts() API.

Returned Values:

This function returns a value 0, if successful.
On error, following value is returned:
WIP_NET_ERR_OPTION: Incorrect configuration parameter entered
WIP_NET_ERR_PARAM: Incorrect configuration parameter value entered

3. Sample Code

```
void adl_main ( adl_InitType_e InitType )
{
            s8 sRet;

   /* Set the option enable IP forwarding to UART1 to */
sRet = wip_netInitOpts(WIP_NET_OPT_IP_FORWARD, TRUE, WIP_NET_OPT_END);
if (sRet != 0)
 TRACE((1, “wip_netInitOpts: Error during init”));
/* Set the TCP/IP protocol options */
sRet = wip_netSetOpts(WIP_NET_OPT_IP_TTL, 10, WIP_NET_OPT_END);
}
```

Summary

The following points have been covered in this section

Internet Plug-In must be initialized before it can be used in a Open AT Application.

Following APIs are available for initialization

- **wip_netInit() to initialize the Internet Plug-In with default options.**
- **wip_netInitOpts() to initialize the Internet Plug-In with user-defined options**
- **wip_netExit() to release the resources allocated during initialization**
- **wip_setOpts() to set the TCP/IP protocol options**
- **wip_getOpts() to get the current TCP/IP protocol options**

CHAPTER 36

IP Stack Bearer Management

Internet Plug-In allows to setup the TCP/IP connection through one of the following bearers:

- PPP client using GSM or UART interface
- PPP server using GSM or UART interface
- GPRS
- Ethernet

Sierra Wireless Internet Plug-in supports multiple IP connection using different bearers. Also in PPP, it allows to use following authentication protocols

- CHAP
- PAP
- MSCHAP v1
- MSCHAP v2

1. Overview

This chapter explains the process that can be used to setup TCP/IP connection using bearer management APIs. It also explains the different APIs available for bearer management.

2. APIs

The header file that should be used for bearer management APIs is "wip_bearer.h". When ethernet bearer is used, "wip_eth.h" should be used.

2.1. Attaching to bearer interface

The wip_bearerOpen() API can be used to attach to bearer interface such as UART, GSM, GPRS.

Prototype:

```
s8 wip_bearerOpen( wip_bearer_t *br, const ascii *device, wip_bearerHandler_f brHdlr, void *context)
```

Parameters:

br: Handle to the bearer
device: Device to attach. It can be one of the following string:
“GSM”
“UART”
“GPRS”
brHdlr: This is the call-back handler, which receives the events related to bearer management service. This handler should be defined as shown below:

void (*wip_bearerHandler_f) (wip_bearer_t br, s8 event, void *context)

Parameters of the call-back handler:
br: Bearer Handle
event: Bearer related events such as authentication failed, IP connected etc.
context: Pointer to application context as defined in wip_bearerOpen() API
context: Pointer to application context. Used to identify a particular instance/or pass application specific data.

Returned Values:

This function returns a value 0, if successful.
On error, following values are returned:
WIP_BERR_NO_DEV: Incorrect device entered
WIP_BERR_ALREADY: Device already used
WIP_BERR_NO_IF: Device not available
WIP_BERR_NO_HDL: Handle not available

2.2. Detaching from bearer interface

The wip_bearerClose() API can be used to detach from bearer interface such as UART, GSM, GPRS.

Prototype:

s8 wip_bearerClose(wip_bearer_t br)

Parameters:

br: Handle to the bearer

Returned Values:

This function returns a value 0, if successful.
On error, following value is returned:
WIP_BERR_BAD_HDL: Invalid handle
WIP_BERR_BAD_STATE: Bearer was not stopped before closing

2.3. Configuration of bearer specific options

The wip_bearerSetOpts() API can be used to configure the bearer specific options such as:
User name
Password

Authentication protocol to use (PAP, CHAP, MSCHAPv1, MSCHAPv2)

Prototype:

s8 wip_bearerSetOpts(wip_bearer_t br, s32 opt, ...)

Parameters:

opt: This specifies the first option in the list of options.
…: List of options supported by this API. All options must be followed by a variable/constant value.The option/value pair will determine what value does a specific parameter takes.. The options supported by this API are defined in file “wip_bearer.h”. Few of the options are mentioned below:
WIP_BOPT_LOGIN: User name for authentication
WIP_BOPT_PASSWORD: Password for authentication
WIP_BOPT_IP_ADDR: Local IP address
WIP_BOPT_IP_DST_ADDR: Remote IP address
WIP_BOPT_GPRS_APN: Access point name
WIP_BOPT_EXTNAT: Enabling the NAT for the interface.
The list of options should be terminated by a macro WIP_BOPT_END.

Returned Values:

This function returns a value 0, if successful.
On error following values are returned:
WIP_BERR_BAD_HDL: Invalid handle
WIP_BERR_OPTION: Invalid option
WIP_BERR_PARAM: Invalid option value

2.4. Retrieving the configuration of bearer specific options

The wip_bearerGetOpts() API can be used to get the current configuration of the bearer specific options such as:
User name
Password
Authentication protocol to use

Prototype:

s8 wip_bearerGetOpts(wip_bearer_t br, s32 opt, ...)

Parameters:

br: bearer handle returned by wip_bearerOpen() API.
opt: This specifies the first option in the list of options.
…: List of options supported by this API. All options must be followed by a variable. In case of get operation, value will be copied to the variable. The options supported by this API are defined in “file wip_bearer.h”. Few of the important options are mentioned below:
WIP_BOPT_LOGIN: User name for authentication
WIP_BOPT_PASSWORD: Password for authentication

WIP_BOPT_IP_ADDR: Local IP address
WIP_BOPT_IP_DST_ADDR: Remote IP address
WIP_BOPT_GPRS_APN: Access point name
WIP_BOPT_EXTNAT: Enabling the NAT for the interface.
The list of options should be terminated by a macro WIP_BOPT_END.

Returned Values:

This function returns a value 0, if successful.
On error following values are returned:
WIP_BERR_BAD_HDL: Invalid handle
WIP_BERR_OPTION: Invalid option

2.5. Establishing bearer connection

The wip_bearerStart() API can be used to establish the bearer connection using bearer interface such as UART, GSM, GPRS.

Prototype:

s8 wip_bearerStart(wip_bearer_t br)

Parameters:

br: Handle to the bearer

Returned Values:

This function returns a value 0, if successful.
On error, following values are returned:
WIP_BERR_BAD_HDL: Invalid handle
WIP_BERR_OK_INPROGRESS: Connection is in progress. An event will be sent on successful establishment connection/reception of error.
WIP_BERR_BAD_STATE: The bearer is already started.
WIP_BERR_DEV: Error from link layer initialization.

2.6. Accepting an incoming PPP connection

The incoming PPP connection can be accepted automatically or it can be accepted manually using the wip_bearerAnswer() API.

Prototype:

s8 wip_bearerAnswer(wip_bearer_t br, wip_bearerServerHandler_f brSrvHdlr, void *context)

Parameters:

br: Handle to the bearer

brSrvHdlr: This is the call-back handler, which receives the events related to PPP server. This handler should be defined as shown below:

```
  s8 (*wip_bearerServerHandler_f)( wip_bearer_t br,wip_bearerServerEvent_t
  *event, void *context)
```

Parameters:

br: Bearer Handle
event: PPP server related events
context: Pointer to application context as defined in wip_bearerAnswer() API

Returned Values:

A positive value should be returned from the handler, if the call has to be established else call will be rejected.
context: Pointer to application context.

Returned Values:

This function returns a value 0, if successful.
On error, following values are returned:
WIP_BERR_BAD_HDL: Invalid handle
WIP_BERR_BAD_STATE: Bearer connection is not closed yet
WIP_BERR_NOT_SUPPORTED: Bearer used is not “GSM”
WIP_BERR_DEV: Error during link layer initialization

2.7. Starting a PPP server

wip_bearerStartServer() API can be used to start bearer in PPP server mode.

Prototype:

```
s8 wip_bearerStartServer( wip_bearer_t br, wip_bearerServerHandler_f brSrvHdlr, void *context)
```

Parameters:

br: Handle to the bearer
brSrvHdlr: This is the call-back handler, which receives the events related to PPP server. This handler should be defined as shown below:

```
  s8 (*wip_bearerServerHandler_f)( wip_bearer_t br,
  wip_bearerServerEvent_t  *event, void *context)
```

Parameters:

br: Bearer Handle
event: PPP server related events. These events include
WIP_BEV_DIAL_CALL: This event signal an incoming call. The call-back function must return a positive value to accept this call.
WIP_BEV_PPP_AUTH_PEER: This signal a PPP authentication request. The call-back function must fill the password (secret) field and should return a positive value to accept the request.
context: Pointer to application context as defined in wip_bearerStartServer() API

Returned Values:

A positive value should be returned from the handler, if the call has to be established else call will be rejected.
context: Pointer to application context.

Returned Values:

This function returns a value 0, if successful.
On error, following values are returned:
WIP_BERR_BAD_HDL: Invalid handle
WIP_BERR_BAD_STATE: Bearer connection is not closed yet
WIP_BERR_NOT_SUPPORTED: Bearer used is not "GSM"
WIP_BERR_DEV: Error during link layer initialization

2.8. Terminating the bearer connection

The wip_bearerStop() API can be used to terminate the bearer connection.

Prototype:

```
s8 wip_bearerStop( wip_bearer_t br)
```

Parameters:

br: Handle to the bearer

Returned Values:

This function returns a value **0**, if successful.
On error, following values are returned:
WIP_BERR_BAD_HDL: Invalid handle
WIP_BERR_OK_INPROGRESS: Connection is in progress. An event will be sent on completion of connection.

2.9. Retrieving the list of available bearers

The wip_bearerGetList() API can be used to get the list of available bearers.

Prototype:

```
wip_bearerInfo_t *wip_bearerGetList( void)
```

Parameters:

None

Returned Values:

This function returns a list of available bearers, if successful.
On error, a NULL pointer is returned.

2.10. Freeing the list of available bearers

The wip_bearerFreeList() API can be used to release the memory allocated to get the list of available bearers.

Prototype:

void wip_bearerFreeList(wip_bearerInfo_t *binfo)

Parameters:

binfo: Pointer to the list of available bearers.

Returned Values:

None

3. Activation of TCP/IP connection

The following section describes the steps which should be taken in the Open AT Application to establish a connection with the service provider. The steps enlisted here are the preliminary steps which should be performed in order to activate the TCP/IP connection between the embedded module and the service provider. Only after activating the TCP/IP connection can other protocols (like TCP/UDP, PING etc) be used. The steps outlined here are indispensable part of any application that utilizes TCP/IP connectivity using Internet Plug-In. These steps are to be performed after the initialization of Internet Plug-in.

3.1. TCP/IP Connection Using GPRS

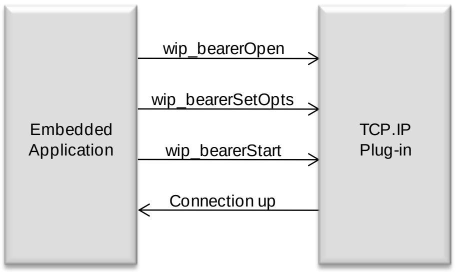


Figure 91 - Connection Using GPRS

In this mode the Sierra Wireless embedded module can connect to TCP/IP network over GPRS. Following are the steps to setup an TCP/IP connection over GPRS:

Step 1: Attachment to GPRS interface

wip_bearerOpen API can be used to select the GPRS interface for TCP/IP connection. Following parameters could be provided during attachment:
Bearer identifier
Interface to use for attachment (GPRS)
Bearer call-back handler, where connection related events will be received

Bearer context to identify particular attachment

For e.g.

```
Ret_Open = wip_bearerOpen(&Br_Id,"GPRS",Bearer_Handler,NULL);
```

Step 2: Configuration of GPRS parameters

After the GPRS network is successfully attached, various GPRS parameters should be set using the API wip_bearerSetOpts(). The parameters that can be configured using this API are:
Access point name
User Name
Password

For e.g.

```
Ret_Option = wip_bearerSetOpts(Br_Id,WIP_BOPT_GPRS_APN,APNServ,
                                    WIP_BOPT_LOGIN,APNUserName,
                                                   WIP_BOPT_PASSWORD,APNPassword,
                                                   WIP_BOPT_END);
```

Step 3: Establishing a TCP/IP connection

wip_bearerStart() API can be used to start the TCP/IP connection. Internet Plug-in does the following during setup of the connection:
Establishes a session with GPRS network
Sends the authentication information such as user name and password
Informs Open AT Application (call back function) that connection setup is successful/failed

For e.g.

```
Ret_Start =wip_bearerStart(Br_Id);
```

3.2. TCP/IP Connection Using PPP

PPP client initiates the PPP session using an outgoing GSM data call or UART interface. The PPP server accepts and asks for authentication information such as username and password. In response, PPP client sends the authentication information over GSM data call or UART interface. If the authentication is successful, PPP server allocates an IP address to the PPP client.

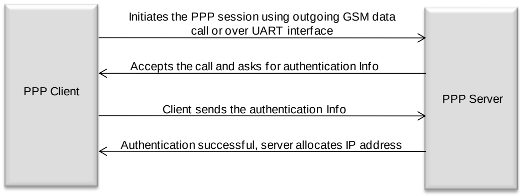

PPP Client
PPP Server
Initiates the PPP session using outgoing GSM data call or over UART interface
Accepts the call and asks for authentication Info
Client sends the authentication Info
Authentication successful, server allocates IP address

Figure 92– TCP/IP Connection Using PPP

Internet Plug-in internally manages the connection between PPP server and client. Internet Plug-in provides APIs to configure the PPP session parameters and start/stop a PPP session.
Sierra Wireless Internet Plug-in allows Sierra Wireless embedded module to behave as a
PPP client
PPP server

3.3. Sierra Wireless embedded module acting as a PPP client

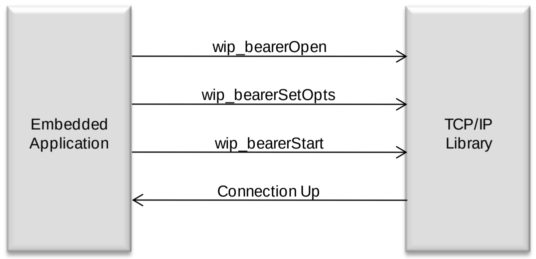


Figure 93 - PPP Client Configuration

In this mode the Sierra Wireless embedded module can connect to
An ISP or
Another GSM embedded module or
Device(PC) connected over UART
to setup a private network.
Following are the steps to setup an TCP/IP connection over PPP when Sierra Wireless embedded module is PPP client:

Step 1: Attachment to bearer interface

Sierra Wireless Internet Plug-in allows to use GSM or UART interface to setup a PPP session. The first step during TCP/IP connection is to choose the interface that should be used for connection.
wip_bearerOpen API can be used to select the interface for TCP/IP connection. Following parameters could be provided during attachment:
Bearer identifier
Interface to use for attachment (UART or GSM)
Bearer call-back handler, where connection related events will be received
Bearer context to identify particular attachment

For e.g.

```
s8 sRet;
wip_bearer_t appBearer;
ascii DeviceName[]="GSM";
/* Bearer connection handler */
void appBearerHandler( wip_bearer_t br, s8 event, void *context)
{
}
void appBearerConnection()
{
                    appBrCxt = adl_memGet(16);
```

```
                wm_strcpy((char *) appBrCxt, "Br Context");
                /* Open the bearer */
                sRet = wip_bearerOpen( &appBearer,
                                                DeviceName,
                                                appBearerHandler,
                                                (void *) appBrCxt);
                TRACE (( 1, "wip_bearerOpen: %d", sRet ));
}
```

Step 2: Configuration of bearer specific parameters

Bearer specific parameters specify the configuration of a client. This includes
Server MSISDN number, user name and password for GSM bearer
User name and password for UART bearer
These parameters can be configured using wip_bearerSetOpts() API.

For e.g.

```
                /* Configure the GSM bearer */
                sRet = wip_bearerSetOpts( appBearer,
                                                WIP_BOPT_DIAL_PHONENB, "123",
                                                WIP_BOPT_DIAL_REDIALCOUNT, 2,
                                                WIP_BOPT_DIAL_REDIALDELAY, 2,
                                                WIP_BOPT_DIAL_RINGCOUNT, 2,
                                                WIP_BOPT_LOGIN, "Un",
                                                WIP_BOPT_PASSWORD, "Pw",
                                WIP_BOPT_END);
```

Step 3: Establishing a TCP/IP connection

wip_bearerStart() API should be used to start the connection. Internet Plug-in does the following during setup of the connection:
In case of GSM
Setups a GSM data call with MSISDN number of PPP server
When the call is setup, sends the authentication information
Informs Open AT Application (call back) that connection setup is successful/failed.
In case of UART
Setups a data connection with the PPP server
When the connection is setup, sends the authentication information
Informs Open AT Application (call back) that connection setup is successful/failed.

For e.g.

```
/* Bearer connection handler */
void appBearerHandler( wip_bearer_t br, s8 event, void *context)
{
        s8 sRet;
        TRACE (( 1, "appBearerHandler: Event %d", event ));
        switch(event)
        {
```

```
                    case WIP_BEV_IP_CONNECTED:
                    {
                            ascii Buffer[50];
                            TRACE (( 1, " WIP_BEV_IP_CONNECTED "));
                            sRet = wip_bearerGetOpts(appBearer,
                                                            WIP_BOPT_IP_ADDR,         &appIpAddr,
WIP_BOPT_END);
                            TRACE (( 1, "wip_bearerGetOpts: %d", sRet));
                            wip_inet_ntoa(appIpAddr , IpAddr, 15);

                            TRACE (( 1, "IP address==>"));
                            TRACE (( 1, IpAddr));
                            wm_sprintf(Buffer, "IP Address==> %s\r\n", IpAddr);
                            adl_atSendResponsePort(ADL_AT_RSP, ADL_PORT_UART1, Buffer);

                    }
                    break;
            default:
                    break;
            }
}
void appBearerConnection()
{
.....
/* Establish the PPP connection */
sRet = wip_bearerStart( appBearer );
TRACE (( 1, "wip_bearerStart: %d", sRet ));
}
```

3.4. Sierra Wireless embedded module acting as a PPP server

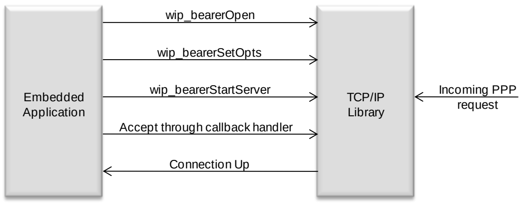


Figure 94 - Configuration for PPP server

This setup can be used when a private TCP/IP network needs to be established between two embedded module or two devices. The Sierra Wireless embedded module in this setup acts as a PPP server which allocates IP address to the PPP client. Following are the steps to setup an TCP/IP connection over PPP when Sierra Wireless embedded module is a PPP server.

Step 1: Attachment to bearer interface

Sierra Wireless Internet Plug-in allows to use GSM or UART interface to setup a PPP session. The first step during TCP/IP connection is to choose the interface that should be used for connection.

wip_bearerOpen API can be used to select the interface for TCP/IP connection. Following parameters could be provided during attachment:
Bearer identifier
Interface to use for attachment (UART or GSM)
Bearer call-back handler where connection related events will be received
Bearer context to identify particular attachment

For e.g.

```
s8 sRet;
wip_bearer_t appBearer;
ascii DeviceName[]="GSM";
/* Bearer connection handler */
void appBearerHandler( wip_bearer_t br, s8 event, void *context)
{
}
void appBearerConnection()
{
                    appBrCxt = adl_memGet(16);
                    wm_strcpy((char *) appBrCxt, "Br Context");
                    /* Open the bearer */
                    sRet = wip_bearerOpen( &appBearer,
                                                        DeviceName,
                                                        appBearerHandler,
                                                        (void *) appBrCxt);
                    TRACE (( 1, "wip_bearerOpen: %d", sRet ));
}
```

Step 2: Configuration of bearer specific parameters

Bearer specific parameters specify the configuration of a server. wip_bearerSetOpts() API can be used for this. The parameters that can be configured using this API are:
Local IP address
IP address that should be allocated to the PPP client
DNS server information

For e.g.

```
void appBearerConnection()
{
          wip_inet_aton( "10.1.1.1", &InAddr);
          wip_inet_aton( "10.1.1.2", &DstAddr);
          /* Configure the bearer */
          sRet = wip_bearerSetOpts( appBearer,
                                              WIP_BOPT_IP_SETDNS,   FALSE,
                                              WIP_BOPT_IP_SETGW,    FALSE,
                                              WIP_BOPT_RESTART,     FALSE,
                                              WIP_BOPT_IP_ADDR, InAddr,
                                              WIP_BOPT_IP_DST_ADDR, DstAddr,
                                              WIP_BOPT_END);
          TRACE (( 1, "wip_bearerSetOpts: %d", sRet ));
}
```

Step 3: Establishing a TCP/IP connection

You can use wip_bearerStartServer() API to configure the embedded module in PPP server mode. When an incoming connection request is received from PPP client, call-back function can be used to accept the incoming request automatically. Alternatively wip_bearerAnswer() API can be used to accept the call manually.
Internet Plug-in does the following during setup of the connection:
Accepts the incoming data call or incoming UART connection
When the data connection is setup, asks for the authentication information from PPP client side
Ask for authentication information from PPP server side
Verifies the PPP client and server information
Allocates IP address to client
Informs Open AT Application (call back) that connection setup is successful/failed.

For e.g.

```
/* PPP Server Handler */
s8 appPPPServHandler( wip_bearer_t br, wip_bearerServerEvent_t *event, void *context)
{
            TRACE (( 1, "appPPPServHandler: Event %d", event->kind ));

  switch (event->kind)
  {
    case WIP_BEV_PPP_AUTH_PEER:
    {
      //automatic PPS answer for trusted "Un" and "Pw"...
      TRACE (( 1, "WIP_BEV_PPP_AUTH_PEER" ));
             TRACE (( 1, "User name provided " ));
             TRACE (( 1, event->content.ppp_auth.user ));

event->content.ppp_auth.secret  = Pw;
      event->content.ppp_auth.secretlen = wm_strlen(Pw);
    }
    break;

    case WIP_BEV_DIAL_CALL:
    {
      TRACE (( 1, "WIP_BEV_DIAL_CALL" ));
      //automatic GSM answer for trusted "PhoneNb"...
      event->content.dial_call.phonenb = PhoneNb;
    }
    break;
  }
            return TRUE;
}
void appBearerConnection()
{
……
                        /* Start the PPP server */
                        sRet = wip_bearerStartServer( appBearer, appPPPServHandler, (void *) NULL);
                        TRACE (( 1, "wip_bearerStartServer: %d", sRet ));
}
```

Summary

The following points have been covered in this section

- **Bearer management is done through one of the following bearers**
    - **GSM**
    - **UART**
    - **GPRS**
- **Sierra Wireless embedded module can be connected to TCP/IP network using**
    - **PPP client using GSM or UART**
    - **PPP Server using GSM or UART**
    - **GPRS**
- **Steps to establish TCP/IP connection using different bearers**

CHAPTER 37

IP Stack Channels

1. Objective

This chapter explains the concept of channel and also the associated APIs

2. Introduction

To manage the layered protocol structure of TCP/IP protocol suite, Internet Plug-In uses a concept of "channel". A channel is an abstract entity which opens a medium over which communication could occur. For e.g. TCP socket channel allows user to transfer data to the remote socket using TCP protocol. All protocols in Internet Plug-In are based on "channel" concept. These channel include:

- TCP server channel
- TCP and UDP socket channel
- Ping channel

3. Architecture

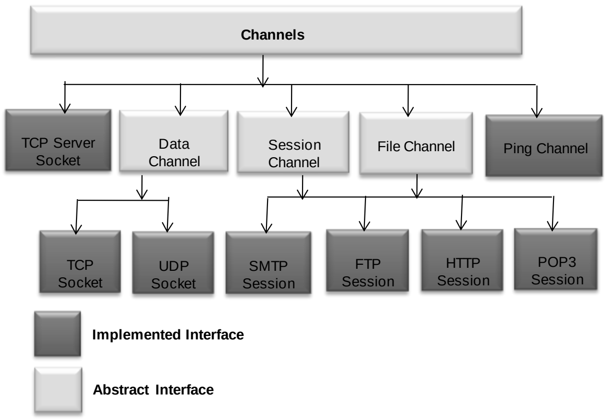


Figure 95 - Channel Architecture

In Internet Plug-In, the communication is achieved through channels. Channels are further inherited by "DataChannel". DataChannel has all properties of "Channels" and additionally it has functionality to transfer data. Currently Internet Plug-In supports only two DataChannels

- TCP Socket

- UDP Socket

TCP server cannot be used to transfer data. To transfer data, it creates a local TCP client socket. This process of creating local socket is referred as “spawning”. A channel has following method or properties

- Open/Close method: To create or close a channel
- Event: Indicates the result or initiation of an action. For e.g. PEER_CLOSE event indicates that channel is closed.
- Event handler: Call-back function that is invoked when an event is received (e.g. READ event on reception of data)
- Data transfer method (Only for DataChannel): Read and Write methods used for transferring data.

4. Read/Write Events

Following section describes when and how the Read/Write events are generated.

4.1. Read Events

The reading of data from the data channel involves the generation of WIP_CEV_READ event by the WIP Library. The WIP_CEV_READ event is ideally generated whenever there is some data to be read in the buffer.

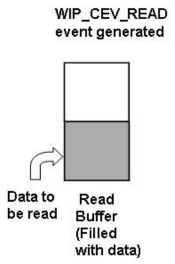


Figure 96 – Read Event Generation

When some data is received in the buffer, a WIP_CEV_READ event is generated to read the data from the buffer as depicted in the figure 8. Note that the read event generation indicates the application to read the data from the buffer using wip_read() API.
For bandwidth optimization over the air, the parameter WIP_COPT_RCV_LOWAT can be configured.
Consider the scenario where the application receives data in chunks. But the requirement is to read the actual data packet size which is received from the remote end.
E.g. Let us consider that a data packet of size 500 bytes is sent over TCP from the server socket and the client socket actually receives it in 5 chunks of 100 bytes each. The requirement is to read only 500 bytes at a time and not chunks of 100 bytes. This necessitates the application to buffer the data and read only a single chunk of 500 bytes. This could be achieved by setting a parameter WIP_COPT_RCV_LOWAT (i.e. set to 499) and reading the required amount of data (i.e. 500). The read event will be received:
First time if there is more than WIP_COPT_RCV_LOWAT bytes to read in the socket’s read buffer
When read attempt returns less data than the requested data and there is more than WIP_COPT_RCV_LOWAT bytes available in the buffer
An added advantage of using the parameter WIP_COPT_RCV_LOWAT includes optimization of bandwidth over the air. For example, we know that an IP packet includes an overhead. But the same IP packet if received in chunks will result in a comparatively higher overhead which would in turn affect the bandwidth utilization.
Let’s consider an example,

WIP_COPT_RCV_BUFSIZE (MAX) has been set to 5840 bytes and WIP_COPT_RCV_LOWAT (MIN) has been set to 1000 bytes.

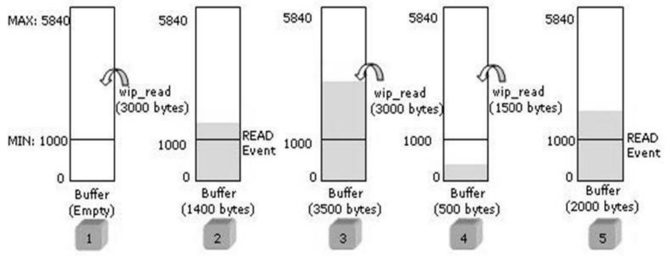


Figure 97 - Generation of Read Events

In this example, the diagram shown above explains the scenario when READ events are received:

Step 1: Attempt is made to read data (3000 bytes).The buffer is empty as data has not been received, so no READ event is received and read will fail.

Step 2: Received 1400 bytes of data in the buffer. In this case, READ event will be received as the size of readable data in the buffer is more than WIP_COPT_RCV_LOWAT, and no READ event has been sent since the last unsuccessful attempt to read.

Step 3: More data (2100 bytes) is received in the buffer. In this case, READ event will not be received, as READ event was already received in Step 2. Data is read (3000 bytes) from the buffer. Size of readable data in the buffer is 500 bytes.

Step 4: Data is read (1500 bytes) from the buffer. Read attempt reads (500 bytes) less data than the requested data, as the available data in the buffer is less.

Step 5: More data (2000 bytes) is received in the buffer. In this case, since the size of the readable data in the buffer (2000 bytes) is more than WIP_COPT_RCV_LOWAT, and there has been an incomplete read (at step 4) since last time a READ event has been received, a new READ event will be received.

4.2. Write Events

The writing of data to the data channel depends on the generation of WIP_CEV_WRITE event by the WIP Library. The write event is ideally generated whenever there is some space in the buffer to write data.

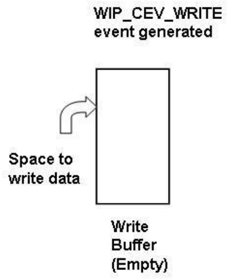


Figure 98 – Scenarios for write event generation

Initially when the buffer is empty, WIP_CEV_WRITE event is generated as depicted in the figure 10. The WIP_CEV_WRITE event is not generated when the buffer has no space to write data. Note that the write event generation indicates the application to write the data to the buffer using wip_write() API.
For bandwidth optimization over the air, the parameter WIP_COPT_SND_LOWAT can be configured.

Write event will be received when:.
Channel is opened for the first time
Write attempt writes less data than the requested data and there are more than WIP_COPT_SND_LOWAT bytes available in the buffer
Let's consider an example,
WIP_COPT_SND_BUFSIZE (MAX) has been set to 5840 bytes and WIP_COPT_SND_LOWAT (MIN) has been set to 1000 bytes.

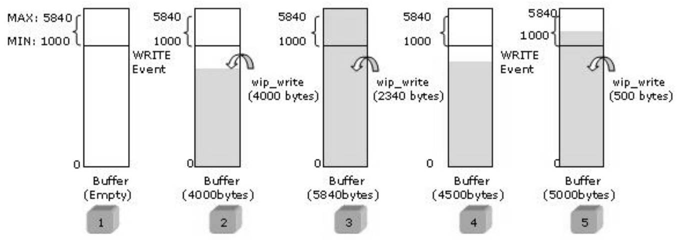


Figure 99 - Generation of Write Events

In this example, the diagram shown above explains the scenario when WRITE events are received:

Step 1: WRITE event is received as the channel is opened for the first time and the buffer is empty.

Step 2: 4000 bytes of data are written to the buffer. In this case, WRITE event will not be received as there is still memory (1840 bytes) to write more data

Step 3: Attempt is made to write data (2340 bytes) more than available buffer size. In this case, only 1840 bytes of data is written successfully to the buffer as the free buffer size is 1840 bytes. Remaining data (500 bytes) will be written to the buffer when the free buffer size becomes equal or more than WIP_COPT_SND_LOWAT.

Step 4: Data is flushed (1340 bytes) from the buffer and now the free buffer is 1340 bytes. In this case, WRITE event will be received, as the free buffer is more than WIP_COPT_SND_LOWAT and there has been no WRITE event since last time a WRITE event has been received.

Step 5: Remaining data (500 bytes) is written to the buffer. In this case, WRITE event will not be received, as there is still memory (840 bytes) to write more data.

5. APIs

5.1. Closing channel

Internet Plug-In provide an API wip_close() to close the channel opened during the TCP/UDP socket communication or while sending a PING request. This API doesn't raise any events.

Prototype:

```
s32 wip_close(wip_channel_t Channel)
```

Parameters:

Channel: The channel you want to close.

Returned Values:

0 is returned on success.
In case of an error, it returns the following:
WIP_CERR_MEMORY: Insufficient memory
WIP_CERR_INVALID: Specified NULL channel

5.2. Finalizers:

Channels generally reserve some heap memory. Depending on their features, it can take some time between the call to wip_close function and the actual releasing of the resources. Although the user might be interested into knowing when a channel closing procedure has been completed, it cannot be reported as a wip_event_t. Since WIP events are attached to the channel, and by definition, the channel does not exist after its release.The users should not use a wip_channel_t in any way after wip_close function has been called on it. If they do, unspecified problems including reboot and memory corruptions might occur. So in order to let users monitor the completion of a channel closure, most channels can be added with a finalizer.
A finalizer is a function which is called after the channel has been completely closed and all its associated resources are freed. Finalizers are attached to channels with the WIP_COPT_FINALIZER option, either in the wip_xxxCreateOpts function, or in the wip_setOpts function.This option will allow to pass a finalizer function to the channel.

Prototype:

typedef void (*wip_finalizer_f) (void *ctx)
This callback type is wip_finalizer_f, and takes the following as parameters:

Parameters:

ctx: context argument

which was attached to the channel.

NOTE :
It is illegal to try to access the (recently destroyed) channel in the finalizer.

5.3. Changing the context of channel

Internet Plug-In provide an API wip_setCtx() to change the context associated with a event handler of the channel. The context of a particular channel is assigned at the time of creation of the socket. In case of the TCP server socket, this context can be changed on for spawned client socket. Note that this context is passed to the callback function associated with the channel each time the callback function is called.

Prototype:

void wip_setCtx (wip_channel_t Channel, void *ctx)

Parameters:

Channel: The channel of which you want to change the context.
ctx: Pointer to the application context defined for particular instance.

Returned Values:

None

5.4. Query the state of the channel

Internet Plug-In provide the API wip_getState() to query the state of a particular channel opened. Since channel creation may depends asynchronous process such as DNS query. A channel create process will not immediately assign a channel. This API will help in getting the present status of the channel.

Prototype:

wip_cstate_t wip_getState(wip_channel_t Channel)

Parameters:

Channel: channel to query the state.

Returned Values:

The various state of the channel that can be returned are:
WIP_CSTATE_BUSY: The channel is still not open, it is being initialized.
WIP_CSTATE_READY: Channel is ready.
WIP_CSTATE_TO_CLOSE: Channel is closed.
Reading data from the channel
There are two mechanisms for using the read API to read data from a particular channel.

- **Polling mechanism**
  - Start a cyclic timer of 100 milliseconds.
  - When timer expires, read the data from the channel using wip_read() or wip_readOpts() API. Read using this API till it returns a value 0.
- **Event Driven**
  - Wait for a READ event in the channel call-back handler
  - When the READ event is received, use wip_read() or wip_readOpts() API to read data. Read using this API till it returns a value 0.

NOTE : READ event is not received each time the data is received by Internet Plug-In. It is only sent when the buffer is empty or reached a threshold. The threshold can be set using a Internet Plug-In wip_setOpts() AP

5.5. Reading data from channel without options

Internet Plug-In provide an API to read the data received for a local socket. This data could be sent by the remote socket in case of TCP or UDP.

Prototype:

s32 wip_read (wip_channel_t Channel, void *buffer, u32 buf_len)

Parameters:

Channel: The channel from which that data should be read.
buffer: The buffer where data will be copied.
buf_len: Length of the buffer.

Returned Values:

The number of bytes read from the channel.
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_CSTATE: The channel is not ready to receive data.
WIP_CERR_NOT_SUPPORTED: This channel does not support data read operation.

5.6. Reading data from channel with options

Internet Plug-In provide an API wip_readOpts() to read the data with advanced configurable options. This options include size of receive buffer etc.

Prototype:

s32 wip_readOpts (wip_channel_t Channel, void *buffer, u32 buf_len, ...)

Parameters:

Channel: The channel from which data should be read.
buffer: The buffer where data will be copied.
buf_len: Length of the buffer.
Options: These option are dependent on the type of the channel from which the data is to be read. The options set for the channel should terminate with WIP_COPT_END option.

Returned Values:

The number of bytes read from the channel. If nothing to read, a value 0 is returned.
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_CSTATE: The channel is not ready to read data.
WIP_CERR_NOT_SUPPORTED: This channel does not support data read operation.

WIP_CERR_INVALID: Invalid option

NOTE :
Only DataChannel supports read operation.

5.7. Writing data to the channel

There are two mechanisms for using the write API to write data to a particular channel.

- **Polling mechanism**
    - Start a cyclic timer of 100 milliseconds.
    - When timer expires, write the data to the channel using wip_write() or wip_writeOpts() API. Write using this API till it returns a value 0.
- **Event Driven**
    - Wait for a WRITE event in the channel call-back handler
    - When the WRITE event is received, use wip_write() or wip_writeOpts() API to write data. Write using this API till it returns a value 0.
    -

5.8. Writing data to the channel without options

Internet Plug-In provide an API wip_write() to write the data to the channel. In case of the UDP and TCP socket the data written over the channel will be sent to the peer socket.

Prototype:

```
s32 wip_write (wip_channel_t Channel, void *buffer, u32 buf_len)
```

Parameters:

Channel: The channel where data should be written.
buffer: The buffer to write.
buf_len: Length of the buffer.

Returned Values:

The number of bytes written to the channel.
In case of an error the various negative error codes that are returned are described below:
WIP_CERR_CSTATE: The channel is not ready to accept data.
WIP_CERR_NOT_SUPPORTED: This channel does not support data writing operation.

NOTE : Only DataChannel supports write operation.

5.9. Writing data to the channel with advanced options

Internet Plug-In provide an API wip_writeOpts() to write data to the channel with advanced configurable options. In case of the UDP and TCP socket the data written over the channel will be sent to the peer socket. This API also provide the facility to configure the parameters such as size of the transmission buffer.

Prototype:

```
s32 wip_writeOpts (wip_channel_t Channel, void *buffer, u32 buf_len, ...)
```

Parameters:

Channel: The channel where data should be written.
buffer: The buffer to write.
buf_len: Length of the buffer.
Options: Options are dependent on the type of the channel from which the data is to be written. The options set for the channel should terminate with WIP_COPT_END option.

Returned Values:

The number of bytes written to the channel.
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_CSTATE: The channel is not ready to accept data
WIP_CERR_NOT_SUPPORTED: This channel does not support data write operation.
WIP_CERR_INVALID: Invalid option

5.10. Configuring the channel

Internet Plug-In provide an API wip_setOpts() to configure a channel. The parameters that can be configured for the channel depends on the type of channel (TCP/UDP etc).

Prototype:

```
s32 wip_setOpts(wip_channel_t Channel, ...)
```

Parameters:

Channel: The channel to configure
Options: The options are dependent on the type of channel. For more information on these parameters, refer to the particular channel APIs. For e.g. To find information on UDP related parameters, refer to parameters used for wip_udpCreateOpts() API.

Returned Values:

0 on success
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_NOT_SUPPORTED: This channel does not support this operation.
WIP_CERR_INVALID: Invalid option

5.11. Retrieving the configuration of a channel

Internet Plug-In provide an API wip_getOpts() to retrieve the configuration of a channel. The parameters that can be read are different for TCP and UDP channels.

Prototype:

```
s32 wip_getOpts(wip_channel_t Channel,…)
```

Parameters:

Channel: The channel from where the configuration should be retrieved.
Options: The options are dependent on the type of channel. For more information on these parameters, refer to the particular channel APIs. For e.g. To find information on UDP related parameters, refer to parameters used for wip_udpCreateOpts() API.

Returned Values:

0 on success
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_NOT_SUPPORTED: This channel does not support this operation.
WIP_CERR_INVALID: Invalid option.
WIP_CERR_CSTATE: The channel is not ready for retrieving the configuration. This will be returned either when channel is still initializing or it is already closed.
WIP_CERR_NOT_SUPPORTED: This channel does not support this operation.

Summary

The following points have been covered in this section:
Channel is an abstract entity which opens a medium over which communication occurs.
Following channel are provided by the wip library:

- **TCP server**
- **TCP socket (Data channel)**
- **UDP socket (Data channel)**
- **Ping**

Following operation can be done on a channel:

- **Set the paramter of the channel.**
- **Write the data to the channel.**
- **Read the data from a channel.**
- **Change the context of the channel.**
- **Get the present state of the channel.**

CHAPTER 38

TCP and UDP Socket Service

1. Objective

This chapter will introduce you to TCP and UDP Socket services provided by Internet Plug-In and will explain how to use them in your Open AT Application. The following topics are discussed in this chapter:
Preliminary tasks to be performed before opening a TCP/UDP socket.

- TCP Sockets
- Types of TCP sockets.
- Configuring a TCP socket.
- Creating a TCP Socket.
- Transfer of Data
- Terminating a TCP socket.
- UDP Sockets
- Configuring UDP socket.
- Creating a UDP Socket.
- Transfer of Data
- Terminating a UDP socket.

2. Introduction

Sierra Wireless Internet Plug-In allows to transfer data using TCP/UDP sockets.

3. Preliminary tasks to be performed before opening a TCP/UDP Socket

TCP/UDP sockets require a physical layer which would actually transfer the data from one peer to another. As you have a embedded module, the physical layer services will be provided by either GPRS network or PPP protocol over UART or GSM. Hence, before opening a TCP/UDP socket, you have to perform certain tasks like initializing Internet Plug-In and attaching to GPRS network or connecting to GSM service provider (in case dialup connection for GSM data call is used). Refer to bearer management chapter for more details.

4. TCP sockets

TCP socket service provides you the capability of exchanging data with any TCP peer in the world. This opens a new mode of communication for your embedded module. As your embedded module becomes internet enabled, you can do a multitude of operations like:

You can open a client/server TCP socket and can exchange status information with HTTP hosts. Hence, you can create web sites which could be used to get data/status information from the embedded module which can be located anywhere in the world.

You can also write Open AT Application to change embedded module settings using the data obtained from HTTP hosts. In this way, you can manage/configure your embedded module by sitting in your home/office using a friendly web interface.

You can create client/server Open AT Application on different embedded module which can then exchange information over the established TCP sockets.

In addition to this, you can implement protocols like HTTP in your Open AT Application and can directly send HTTP requests (Get/Post) from your Open AT Application. This increases the power of your Open AT Application.
You do not have to worry about the implementation of TCP socket communication as the Internet Plug-In handles all the internal intricacies.

5. Types of TCP sockets

There are two types of TCP sockets which you can open and use to exchange data. These are:

- TCP server socket.
- TCP client socket.

5.1. TCP server socket

When a server socket is created using Internet Plug-In, socket passively listens on a specified port for incoming connections. The below mentioned diagram shows different states managed for TCP server.

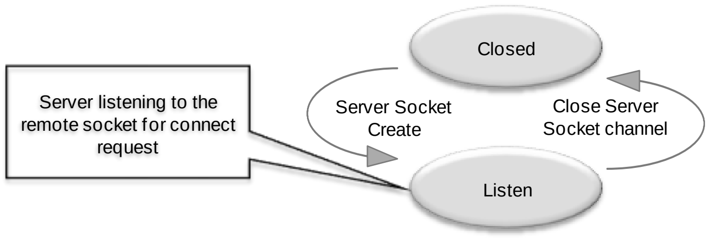

Closed
Server Socket Create
Close Server Socket channel
Server listening to the remote socket for connect request
Listen

Figure 100 - State diagram of the TCP server socket

On reception of a connection request from a remote client socket, a server socket does the following,
Spawns a new socket (client) to connect to the remote socket
Data transfer is done between the spawned socket and the remote socket
Server socket remains in the listening mode and is ready to accept the request from other clients.

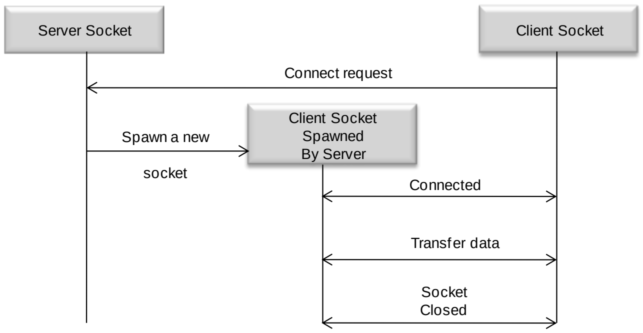


Figure 101 - Connection establishment procedure

5.2. TCP client socket

Client socket to connect to a server TCP socket and exchange data with it. Client socket can be created in two ways using Internet Plug-In:
Client socket created using Internet Plug-In API directly
Client socket spawned by the server socket

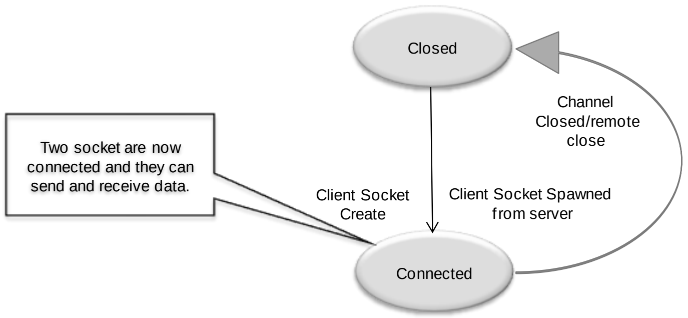

Closed
Channel Closed/remote close
Two socket are now connected and they can send and receive data.
Client Socket Create
Client Socket Spawned from server
Connected

Figure 102 - State diagram of the TCP client socket

6. Include file

To use the TCP service include the following header file:
wip_tcp.h

6.1. Creating a TCP server without options

Internet Plug-In provide an API wip_TCPServerCreate() to create a TCP server with default options. When a connection request is received from remote socket, server spawns the client socket.

Prototype:

```
wip_channel_t wip_TCPServerCreate ( u16 Port, wip_eventHandler_f commHandler, void *ctx)
```

Parameters:

Port: This parameter indicates the port on which the TCP server will listen for incoming connections.

Range: 1 to 65535.
Type: u16
commHandler: The event handler for client TCP sockets. This handler reacts to events corresponding to the spawned client sockets and not corresponding to the server socket.

Prototype:

```
void (*wip_eventHandler_f) ( wip_channel_t Channel,
                             wip_event_t Event, void *ctx);
```

Parameters of the call-back handler:

Channel: Defines the communication channel corresponding to the client socket spawned by the server.
Event: Structure indicating the event corresponding to the channel. It also carries the data related to the corresponding event. The various event and corresponding data that are notified are:

WIP_CEV_OPEN: Connection is established with the remote server. This event is sent when the spawned socket accepts the connection request sent by the remote socket. In case of the client socket this event is generated, when server accepts the connection request.
WIP_CEV_READ: This event signifies that the data is being received from the remote socket and can be read using the read API.
WIP_CEV_WRITE: This event signifies that Internet Plug-In is ready to send data.
WIP_CEV_ERROR: This event is sent in case of an error
WIP_CEV_PEER_CLOSE: This event is sent when socket is closed.

ctx: Pointer to application context for a particular instance.
ctx: Pointer to application context for a particular instance.

Returned Values:

Created channel ID
NULL on error

NOTE :
The channel ID assigned to the socket spawned by the server is notified by the event structure.
Each WIP_CEV_WRITE and WIP_CEV_READ event will notify the channel to which the data can be written to or read from.

6.2. Starting TCP server with options

Internet Plug-In provide the API wip_TCPServerCreateOpts() to create the socket with the user defined parameters. This API allows you to configure the various advanced parameters such as transmission buffer size, receive buffer size etc.

Prototype:

```
wip_channel_t wip_TCPServerCreateOpts( u16 Port, wip_eventHandler_f commHandler, void
*ctx, ...)
```

Parameters:

Port: This parameter indicates the port on which the TCP server will listen for incoming connections.
Range: 1 to 65535.
Type: u16

commHandler: The event handler for client TCP sockets. This handler reacts to events corresponding to the spawned client sockets and not corresponding to the server socket. Refer to section: 6.1 for more details.
ctx: Pointer to application context for a particular instance.
List of options supported by this API. All options must be followed by a variable. In case of get operation, value will be copied to the variable while in case of set operation, value will be taken from the variable. The options supported by this API are:

WIP_COPT_END: To mark the termination of the parameter list.
WIP_COPT_SND_BUFSIZE: The size of the transmission buffer for the spawned socket.
Type: u32
WIP_COPT_RCV_BUFSIZE: The size of the receive buffer for the spawned socket.
Type: u32
WIP_COPT_SND_LOWAT: This parameter tells the minimum amount of space that should be available in the transmission buffer. After the buffer is full, it triggers a WIP_CEV_WRITE event.
Type: u32.
WIP_COPT_RCV_LOWAT: This parameter signifies the minimum amount of space that should be available in the receive buffer before it triggers WIP_CEV_READ event.
Type: u32
WIP_COPT_SND_TIMEOUT: Data is sent to the remote socket when the buffer is full or in case a particular time (time out period for transmission buffer) has elapsed before the buffer is full. This parameter is used to set the value of this time out period.
Type: u32
WIP_COPT_NODELAY: If this flag is set to true then data is sent immediately.
Type: bool
WIP_COPT_TOS: Type of service. Please refer to RFC 791 for more details.
Type: u8
WIP_COPT_TTL: The time for which data is valid.
Type: u8
WIP_COPT_FINALIZER: Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.
Type: wip_finalizer_f
WIP_COPT_REXMT_MAX: Sets the maximum time between TCP retransmissions.
Type: u32
WIP_COPT_REXMT_MAXCNT: Sets the maximum number of retransmissions.
Type: u32

Returned Values:

Created channel ID
NULL on error

6.3. Sample Code for TCP Server

```
#include "wip.h"
const u16 wm_apmCustomStackSize = 1024;

/* Setting for GPRS */
ascii APNServ[]="www";
ascii APNUserName[]="";
ascii APNPassword[]="";
```

```
/*Identifer for the bearer*/
wip_bearer_t Br_Id;

/* Channel identifier */
wip_channel_t Channel; /*Channel for the Server*/
wip_channel_t Client_Channel; /*Channel ID for the Client spawned*/

/*Control Handler for the Client socket spawned*/
void TCP_Handler(wip_event_t *Event,void *ctx)
{
        u32 Read;
        ascii *Data_Buffer;
        u32 Write;
        TRACE (( 1, "Inside TCP_Handler"));
        Client_Channel = Event->channel;
        switch(Event->kind)
        {
        case WIP_CEV_READ:
                Data_Buffer        = adl_memGet((Event->content.read.readable+1));
                wm_memset(Data_Buffer,0x00,Event->content.read.readable);
                TRACE (( 1, "Inside WIP_CEV_READ"));
                /*Read the data send by the remote party */
                Read = wip_read(Client_Channel,Data_Buffer, Event->content.read.readable);
                TRACE (( 1, "Data recevied: %d",Read));
                *(Data_Buffer+Read) ='\0';
                if(Read>0)
                {
                        /*Send the data read to the external application*/
                        adl_atSendResponsePort(ADL_AT_UNS, ADL_PORT_UART1,Data_Buffer);
                        adl_atSendResponsePort(ADL_AT_UNS, ADL_PORT_UART1,"\r\n");
                }
                _adl_memRelease(&Data_Buffer);
                break;
        case WIP_CEV_WRITE:
                TRACE (( 1, "Inside WIP_CEV_WRITE"));
                break;
  case WIP_CEV_OPEN:
                TRACE (( 1, "Inside WIP_CEV_OPEN"));
                /*Client is conneted to the server, send the test data*/
                Write = wip_write(Client_Channel,"This is a test data",19);
                break;
  case WIP_CEV_PEER_CLOSE:
                TRACE (( 1, "Inside WIP_CEV_PEER_CLOSE"));
                break;
        }
}
/*Handler to receive all the event related to Bearer (GSM,GPRS,UART) */
```

```
void Bearer_Handler(wip_bearer_t *Br,s8 Event, void * Context)
{
        ascii Trace_Buf[100];
        u16 Port =80;
        TRACE (( 1, "Inside Bearer_Handler" ));
        switch(Event)
        {
        case WIP_BEV_IP_CONNECTED:
                TRACE (( 1, " WIP_BEV_IP_CONNECTED "));
                /*Open a TCP server after the client is connected to the bearer*/
                Channel = wip_TCPServerCreate(Port,TCP_Handler,NULL);
                break;
        default:
                wm_sprintf(Trace_Buf,"Value in callback handler for bearer: %d\r\n",Event);
                TRACE (( 1,Trace_Buf));
        }
}
```

6.4. Creating a TCP client without options

Internet Plug-In provide the API wip_TCPClientCreate() to create a TCP client socket with default options.

Prototype:

wip_channel_t wip_TCPClientCreate (const ascii * serverAddr, u16 serverPort, wip_eventHandler_f commHandler, void *ctx)

Parameters:

serverAddr: IP address of the server to which the client has to connect.
Port: This parameter indicates the port of server to which the TCP client socket has to connect.
Range: 1 to 65535.
Type: u16
commHandler: This is the call-back function that notifies the event corresponding the client sockets. Refer to section: 6.1 for more details on commHandler.
ctx: Pointer to the application context.

Returned Values:

Created channel ID
NULL on error

6.5. Creating a TCP client with options

Internet Plug-In provide the API wip_TCPClientCreateOpts() to create a TCP client socket with advanced options such as send and receive buffer size.

Prototype:

wip_channel_t wip_TCPClientCreateOpts(const ascii * serverAddr, u16 serverPort, wip_eventHandler_f commHandler, void * ctx, ...)

Parameters:

serverAddr: IP address of the server to which the client has to connect.
Port: This parameter indicates the port of server to which the TCP client socket has to connect.
Range: 1 to 65535.
Type: u16
commHandler: Call-back function that notifies the event corresponding the client socket. Refer to section 6.1 for more details on commHandler.

ctx: Pointer to the application context.
Options: List of options supported by this API. All options must be followed by a variable. In case of get operation, value will be copied to the variable while in case of set operation, value will be taken from the variable. The options supported by this API are:

WIP_COPT_END: To mark the termination of the parameter list.
WIP_COPT_KEEPALIVE: If this parameter is set, an empty packet is sent by the TCP socket every nth second to keep the TCP socket alive.
Type: u32 – Defines number of seconds
WIP_COPT_SND_BUFSIZE: The size of the transmission buffer for client socket.
Type: u32
WIP_COPT_RCV_BUFSIZE: The size of the receive buffer for the client socket.
Type: u32
WIP_COPT_SND_LOWAT: Minimum amount of space that should be available in the transmission buffer. After the buffer is full, it triggers a WIP_CEV_WRITE event.
Type: u32
WIP_COPT_RCV_LOWAT: This parameter signifies the minimum amount of space that should be available in the receive buffer before it triggers WIP_CEV_READ event.
Type: u32
WIP_COPT_NODELAY: If this flag is set, then data is sent immediately.
Type: bool
WIP_COPT_MAXSEG: Maximum size of the TCP packet.
Type: u32
WIP_COPT_TOS: Type of service require, refer to RFC 791 for more details.
Type: u8
WIP_COPT_TTL: The time for which the data is valid.
Type: u8
WIP_COPT_STRADDR: Local address of the socket.
Type: ascii
WIP_COPT_PORT: Port occupied by the socket.
Type: u16
WIP_COPT_REXMT_MAX: Sets the maximum time between TCP retransmissions.
Type: u32
WIP_COPT_REXMT_MAXCNT: Sets the maximum number of retransmissions.
Type: u32

Returned Values:

Created channel ID
NULL on error

6.6. Sample Code for TCP Client

```
#include "wip.h"
const u16 wm_apmCustomStackSize = 1024;

/* Setting for GPRS */
ascii APNServ[]="www";
ascii APNUserName[]="";
ascii APNPassword[]="";

/*Identifer for the bearer*/
wip_bearer_t Br_Id;

/*Configuration of the server to connect*/
u16 Port= 80;
ascii IP[] = "12.15.7.214";

/*Channel ID for the client socket*/
wip_channel_t Channel;

/*Handles the event related to the client sockets*/
void Client_Event_Handler(wip_event_t *Event, void *ctx)
{
        u32 Read;
        ascii *Data_Buffer;
        u32 Write;
        TRACE (( 1, "Inside TCP_Handler"));

        switch(Event->kind)
        {
        case WIP_CEV_READ:
                TRACE (( 1, "Inside WIP_CEV_READ"));
                Data_Buffer        = adl_memGet((u16) (Event->content.read.readable+1));
                wm_memset(Data_Buffer,0x00,Event->content.read.readable);
                /*Read the data send by the remote party */
                Read = wip_read(Channel,Data_Buffer, Event->content.read.readable);
                *(Data_Buffer+Read) ='\0';
                if(Read>0)
                {
                        /*Send the data read to the external application*/
                        adl_atSendResponsePort(ADL_AT_UNS, ADL_PORT_UART1,Data_Buffer);
                        adl_atSendResponsePort(ADL_AT_UNS, ADL_PORT_UART1,"\r\n");
                }
                else
```

```
                        {
                                TRACE (( 1, "Read Error"));
                        }
                        _adl_memRelease(&Data_Buffer);
                        break;
                case WIP_CEV_OPEN:
                        TRACE (( 1, "Inside WIP_CEV_OPEN"));
                        adl_atSendResponse(ADL_AT_UNS,"Client is connected\r\n");
  break;
                case WIP_CEV_WRITE:
                        TRACE (( 1, "Inside WIP_CEV_WRITE"));
                        Write = wip_write(Channel,"This is a test data",19);
                        break;
                }
}

/*Handler to receive all the event related to Bearer (GSM,GPRS,UART) */
void Bearer_Handler(wip_bearer_t *Br,s8 Event, void * Context)
{
                ascii Trace_Buf[100];
                TRACE (( 1, "Inside Bearer_Handler" ));
                switch(Event)
                {
                case WIP_BEV_IP_CONNECTED:
                        TRACE (( 1, " WIP_BEV_IP_CONNECTED "));
                        /*On connecting to the bearer create a client TCP socket to connect to a
        server*/
        Channel = wip_TCPClientCreate(IP,Port,Client_Event_Handler,NULL);
                default:
                        wm_sprintf(Trace_Buf,"Value in callback handler for bearer: %d\r\n",Event);
                        TRACE (( 1,Trace_Buf));
                }
}
```

6.7. Aborting a TCP socket

Internet Plug-In provide an API wip_abort() to abort channel. The channel will not be closed using this API. Hence channel should be closed using wip_close() when user wants to close the channel. The wip_abort() API aborts the TCP connection and data is not sent anymore on socket. The wip_close() API clears the internal data allocated by Internet Plug-In for this channel.

Prototype:

s32 wip_abort (wip_channel_t Channel)

Parameters:

Channel: The channel ID of the socket which has to be aborted.

Returned Values:

0 on success
In case of an error a negative value is returned.
WIP_CERR_NOT_SUPPORTED: This value is returned in case this API is used to abort either TCP server or UDP channel.
WIP_CERR_INTERNAL: This value is returned when there is a internal problems to abort the channel.

6.8. Shutting TCP communication

Internet Plug-In provide an API wip_shutdown() to shut either the input or output communication of a given socket. If both communications are shut down, the socket is closed. If the output communication is closed, the peer socket is advised by a WIP_CEV_PEER_CLOSE error event.

Prototype:

```
s32 wip_shutdown (wip_channel_t Channel, bool read, bool write)
```

Parameters:

Channel: The channel ID of the socket to be shutdown.
read: If this parameter is set to TRUE, input communication over the socket will closed down. In other words this socket will not be able to receive any data.
write: If this parameter is set to TRUE, then output communication will be closed down for the given channel. In other words this socket will not be able to send any data.

Returned Values:

0 on success
In case of an error a negative value is returned.
WIP_CERR_NOT_SUPPORTED: This value is returned in case this API is used to shutdown either TCP server or UDP channel.
WIP_CERR_INTERNAL: This value is returned when there is a internal problems to shutdown the channel.

7. UDP sockets

Like TCP socket services, UDP socket services too allow you to send and receive data (called UDP datagram) from any UDP entity around the world. The difference between TCP and UDP is that, UDP is a connectionless protocol whereas TCP is connection oriented protocol.
TCP provides reliable delivery of packets whereas UDP does not. In UDP, a packet (datagram) can be transmitted without establishing a connection with the peer (i.e. a client can send a datagram without establishing a connection with UDP server). They do not have a peer socket with which they will exclusively exchange the data. So with each outbound data the destination IP has to be specified.

The below diagram show the data communication between three UDP sockets. All sockets can send and receive data between them.

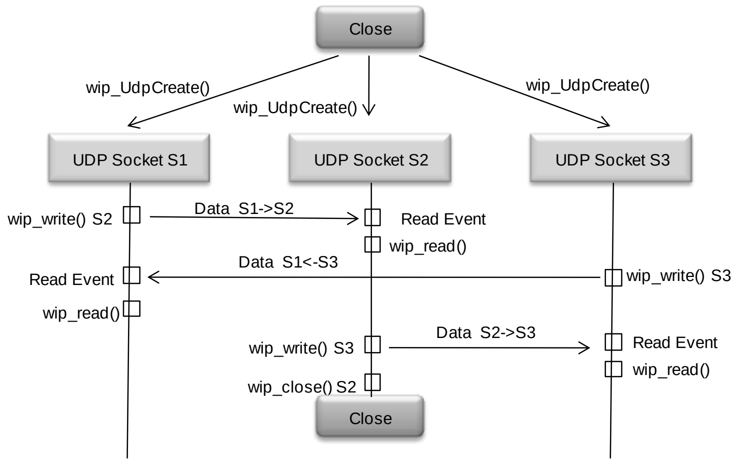


Figure 103 - UDP socket data communication

7.1. Creating a UDP socket without options

Internet Plug-In provide the API wip_UDPCreate() to create a API with the default parameter set in the library for the UDP socket.

Prototype:

wip_channel_t wip_UDPCreate (wip_eventHandler_t commHandler, void *ctx)

Parameters:

commHandler: The event handler attached to the UDP socket. This handler reacts to events corresponding to the UDP.

Prototype:

void (*wip_eventHandler_f) (wip_channel_t Channel, wip_event_t Event, void *ctx)

Parameters of the call-back handler:
Channel: Defines the particular communication channel correspond to the opened UDP socket.
Event: Structure which indicates the type of event corresponding to the channel. This structure also carries the data related to the corresponding event. The various event and corresponding data that are notified are:

WIP_CEV_OPEN: This event is sent whenever the socket is open and ready to send and receive data to the other sockets.
WIP_CEV_READ: This event signifies that the data is received from the remote socket and can be read using the read API.
WIP_CEV_WRITE: This event signifies that Internet Plug-In is ready to send data.
WIP_CEV_ERROR: This event notifies the error corresponding the channel.

ctx: Pointer to the application context as defined in wip_UDPCreate() API
ctx: Pointer to the application context

Returned Values:

Created channel ID
NULL on error

7.2. Creating a UDP socket with options

Internet Plug-In provide the API to create a API with advanced configurable options such as peer port, peer IP address etc.

Prototype:

```
wip_channel_t wip_UDPCreateOpts ( wip_eventHandler_t commHandler,
void *ctx, ...)
```

Parameters:

commHandler: Refer to section 7.1 for details.
ctx: Pointer to the application context.
Options: List of options supported by this API. All options must be followed by a variable. In case of get operation, value will be copied to the variable while in case of set operation, value will be taken from the variable. The options supported by this API are:

WIP_COPT_SND_BUFSIZE: The size of the transmission buffer for UDP socket
Type: u32
WIP_COPT_RCV_BUFSIZE: The size of the receive buffer for the UDP socket.
Type: u32
WIP_COPT_CHECKSUM: If checksum control has to be performed by the UDP socket or not.
Type: bool
WIP_COPT_TOS: Type of service require, refer to RFC 791 for more details.
Type: u8
WIP_COPT_TTL: The time for which the data is valid. The purpose of the TTL field is to avoid a situation in which an undeliverable datagram keeps circulating on an internet system.
Type: u8
WIP_COPT_DONTFRAG: If this parameter is set to true than the UDP datagram is not allowed to be break into fragment through the network.
Type: bool
WIP_COPT_PORT: Port occupied the socket opened.
Type: u16
WIP_COPT_STRADDR: Local address of the server in ascii format.
Type: ascii
WIP_COPT_ADDR: Local address of the server in u32 format
Type: u32
WIP_COPT_PEER_PORT: Port of the peer socket to which the data will be sent.
Type: u16
WIP_COPT_PEER_STRADDR: IP address of the peer socket to which the data is to be sent in ascii format.
Type: ascii
WIP_COPT_PEER_ADDR: IP address of the peer socket to which the data is to be sent in u32 format.

Type: ascii
WIP_COPT_BOUND: Specifies if the socket is bound to a peer socket or not.
Type: boolean
WIP_COPT_FINALIZER: Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.
Type: wip_finalizer_f

Returned Values:

Created channel ID
NULL on error

7.3. Sample Code for UDP

```
/* Open the first UDP channel */
void StartUDP1()
{
        TRACE((1, "UDP Channel1 Creation"));
        UDPChannel1 = wip_UDPCreateOpts(UDPChannel1Hdlr, NULL,
                                        WIP_COPT_PEER_PORT, PEER_PORT1,
                                        WIP_COPT_PEER_STRADDR, PEER_ADDR,
                                        WIP_BOPT_END);
        if (UDPChannel1 != (wip_channel_t)NULL)
                TRACE((1, "UDP Channel Created"));
}
```

Summary

The following points have been covered in this section

- **There are two types of TCP socket**
    - **Server Socket**
    - **Client socket**
- **There are two APIs to open TCP server socket.**
    - **wip_TCPServerCreate (socket with default parameter)**
    - **wip_TCPServerCreateOpts(socket with user defined parameters)**
- **There are two API to open a TCP client socket.**
    - **wip_TCPClientCreate (socket with default parameter)**
    - **wip_TCPClientCreateOpts(socket with user defined parameters)**
- **In order to close the socket the API used is**
    - **wip_abort**
- **In order to close a particular communication(input/output) channel for the socket the API used is**
    - **wip_shutdown**
- **A UDP socket is connection less protocol.**
- **At any time a socket can send and receive data from any peer socket, there is no dedicated communication link between the socket.**
- **In order to open an UDP socket there are two APIs provided.**
    - **wip_UDPCreate (creating a UDP socket with default parameters)**
    - **wip_UDPCreateOpts (creating a UDP socket with user defined parameters)**

CHAPTER 39

Management of IP Address Formats

1. Objective

This chapter explain the APIs available in Internet Plug-In for managing IP address formats.

2. Internet protocol support API

Internet Plug-In provide utilities to manage the IP address. There are two format of the internet address:

- Standard dot format notation format for the internet address for example: 255.255.255.255.
    - Address is represented in a unsigned 32 bit format (network byte order)
    - Internet Plug-In provide APIs to convert either of the format to other.
    - In order to use this service the header file that need to be included is
    - wip_inet.h
- The 32bit format of the address is defined in the library as follows:

```
typedef u32 wip_in_addr_t
```

3. Conversion from dot notation to unsigned format.

Internet Plug-In provide the API to convert the IP address in ascii dot format to u32 format.

Prototype:

```
bool wip_inet_aton(const ascii *str, wip_addr_t *addr)
```

Parameters:

str: String containing the IP address in standard dot notation format.
addr: The function will fill this variable with address in u32 format.

Returned Values:

TRUE is returned if the string contains a valid IP address.
FALSE is returned if the string does not contains a valid address.

4. Conversion from unsigned format to dot notation.

Internet Plug-In provide the API to convert the IP address in u32 format to ascii dot format.

Prototype:

```
bool wip_inet_ntoa (wip_addr_t addr, ascii *buff, u16 buflen)
```

Parameters:

addr: The IP address in u32 format.
buff: Buffer that will be filled with the address in standard dot notation format.
buflen: Length of the buffer.

Returned Values:

TRUE is returned if the provided buffer is long enough to store the IP address
FALSE is returned if buffer is not of sufficient size to store the IP address.

5. Sample Code

```
        case WIP_BEV_IP_CONNECTED:
                TRACE (( 1, " WIP_BEV_IP_CONNECTED "));
                sRet = wip_bearerGetOpts(Br_Id,
                                        WIP_BOPT_IP_ADDR, &appIpAddr, WIP_BOPT_END);
                TRACE (( 1, "wip_bearerGetOpts: %d", sRet));
                wip_inet_ntoa(appIpAddr , IpAddr, 15);

                TRACE (( 1, "IP address==>"));
                TRACE (( 1, IpAddr));
```

Summary

The following points have been covered in this section

- **Internet Plug-In provide the API to convert the IP address in 32 bit format to dot notation format and vice versa.**
- **The API provided for this purpose are:**
    - **wip_inet_aton()**
    - **wip_inet_ntoa()**

CHAPTER 40

File API for IP services

1. Objective

The Internet Plug-In provide APIs for establishing a connection to FTP/HTTP/SMTP/POP3 server.

Create session channel using wip_xxxClientCreate () or wip_xxxClientCreateOpts () for establishing a connection to the server

Create data channel using wip_getFile ()/wip_getFileOpts ()/wip_putFile ()/ wip_putFileOpts() for data transfer.
This chapter explain the APIs available in Internet Plug-In for creating data channel which can be used for data transfer.

2. File APIs

The Internet Plug-In provide APIs for

- Creating data channel
- Manipulating directory
- Setting and retrieving protocol specific options

In order to use this service the header file that need to be included is
wip_file.h

2.1. Creating Data Channel Without Options for Downloading Data

Internet Plug-In provide an API wip_getFile () to create a data channel for downloading data from the server. This API will create the data channel with default parameters.

Prototype:

```
wip_channel_t wip_getFile( wip_channel_t session_ch, ascii *file_name, wip_eventHandler_f evh, void *ctx)
```

Parameters:

session_ch: Connection channel returned by wip_xxxClientCreate() / wip_xxxClientCreateOpts()
file_name: Name of the file to be downloaded from the server

evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully for exchanging data.
- WIP_CEV_READ: This event signifies that the data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event signifies that Internet Plug-In is ready to send data.
- WIP_CEV_ERROR: This event is sent in case of an error
- WIP_CEV_PEER_CLOSE: This event is sent when the complete data is downloaded.

Ctx: Pointer to application context.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

2.2. Creating Data Channel With Options for Downloading Data

Internet Plug-In provide an API wip_getFileOpts () to create a data channel for downloading data from the server. This API will create the data channel with the options specific to protocol.

Prototype:

```
wip_channel_t wip_getFileOpts( wip_channel_t session_ch, ascii *file_name, wip_eventHandler_f evh, void *ctx, ...)
```

Parameters:

session_ch: Connection channel returned by wip_xxxClientCreate() / wip_xxxClientCreateOpts()
file_name: Name of the file to be downloaded from the server
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully for exchanging data.
- WIP_CEV_READ: This event signifies that the data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event signifies that Internet Plug-In is ready to send data.
- WIP_CEV_ERROR: This event is sent in case of an error
- WIP_CEV_PEER_CLOSE: This event is sent when the complete data is downloaded.

Ctx: Pointer to application context.
...: This specifies the option specific to protocol with the option value. Note that the list must be terminated by WIP_COPT_END.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

2.3. Creating Data Channel Without Options for Uploading Data

Internet Plug-In provide an API wip_putFile () to create a data channel for uploading data to the server. This API will create the data channel with default parameters.

Prototype:

wip_channel_t wip_putFile(wip_channel_t session_ch, ascii *file_name, wip_eventHandler_f evh, void *ctx);

Parameters:

session_ch: Connection channel returned by wip_xxxClientCreate() / wip_xxxClientCreateOpts()
file_name: Name of the file to be uploaded to the server
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully for exchanging data.
- WIP_CEV_READ: This event signifies that the data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event signifies that Internet Plug-In is ready to send data.
- WIP_CEV_ERROR: This event is sent in case of an error
- WIP_CEV_PEER_CLOSE: This event is sent when the complete data is downloaded.

Ctx: Pointer to application context.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

2.4. Creating Data Channel With Options for Uploading Data

Internet Plug-In provide an API wip_putFileOpts () to create a data channel for uploading data to the server. This API will create the data channel with the options specific to protocol.

Prototype:

wip_channel_t wip_putFileOpts(wip_channel_t session_ch, ascii *file_name, wip_eventHandler_f evh, void *ctx, ...);

Parameters:

session_ch: Connection channel returned by wip_xxxClientCreate() / wip_xxxClientCreateOpts()
file_name: Name of the file to be uploaded to the server
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully for exchanging data.

- WIP_CEV_READ: This event signifies that the data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event signifies that Internet Plug-In is ready to send data.
- WIP_CEV_ERROR: This event is sent in case of an error
- WIP_CEV_PEER_CLOSE: This event is sent when the complete data is downloaded.

Ctx: Pointer to application context.
...: This specifies the option specific to protocol with the option value. Note that the list must be terminated by WIP_COPT_END.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

2.5. Changing the Working Directory

Internet Plug-In provide an API wip_cwd () to change the current working directory. When this API successfully changes the working directory, WIP_CEV_DONE event will be received in the event handler, otherwise WIP_CEV_ERROR event will be received. During this operation, the channel will be in WIP_CSTATE_BUSY state indicating that the no operation is allowed on this context till one of these events are received.

Prototype:

```
s32 wip_cwd( wip_channel_t session_ch, ascii *directoy_name)
```

Parameters:

session_ch: Connection channel whose directory needs to be changed
directory_name: Name of the new directory

Returned Values:

0 is returned if the directory change request has been sent successfully.
Negative error code is returned on failure.

2.6. Creation of New Directory

Internet Plug-In provide an API wip_mkdir () to create a new directory in the current working directory.
When this API successfully creates the new directory, WIP_CEV_DONE event will be received in the event handler, otherwise WIP_CEV_ERROR event will be received. During this operation, the channel will be in WIP_CSTATE_BUSY state indicating that the no operation is allowed on this context till one of these events are received.

Prototype:

```
s32 wip_mkdir( wip_channel_t session_ch, ascii *directoy_name)
```

Parameters:

session_ch: Connection channel returned by wip_xxxClientCreate() / wip_xxxClientCreateOpts()
directory_name: Name of the new directory that needs to be created

Returned Values:

0 is returned if the new directory creation request has been sent successfully.
Negative error code is returned on failure.

2.7. Deleting a File

Internet Plug-In provide an API wip_deleteFile () to delete a file. When this API successfully deletes a file, WIP_CEV_DONE event will be received in the event handler, otherwise WIP_CEV_ERROR event will be received. During this operation, the channel will be in WIP_CSTATE_BUSY state indicating that the no operation is allowed on this context till one of these events are received.

Prototype:

```
s32 wip_deleteFile( wip_channel_t session_ch, ascii *file_name)
```

Parameters:

session_ch: Connection channel returned by wip_xxxClientCreate() / wip_xxxClientCreateOpts()
file_name: Name of the file to delete

Returned Values:

0 is returned if the file deletion request has been sent successfully.
Negative error code is returned on failure.

2.8. Deleting a Directory

Internet Plug-In provide an API wip_deleteDir () to delete an empty directory. When this API successfully deletes a directory, WIP_CEV_DONE event will be received in the event handler, otherwise WIP_CEV_ERROR event will be received. During this operation, the channel will be in WIP_CSTATE_BUSY state indicating that the no operation is allowed on this context till one of these events are received.

Prototype:

```
s32 wip_deleteDir( wip_channel_t session_ch, ascii *dir_name)
```

Parameters:

session_ch: Connection channel returned by wip_xxxClientCreate() / wip_xxxClientCreateOpts()
dir_name: Name of the directory to delete

Returned Values:

0 is returned if the directory deletion request has been sent successfully.
Negative error code is returned on failure.

2.9. Renaming a File

Internet Plug-In provide an API wip_renameFile () to change the file name. When this API successfully changes the file name, WIP_CEV_DONE event will be received in the event handler, otherwise WIP_CEV_ERROR event will be received. During this operation, the channel will be in WIP_CSTATE_BUSY state indicating that the no operation is allowed on this context till one of these events are received.

Prototype:

```
s32 wip_renameFile( wip_channel_t session_ch, ascii *old_name, ascii *new_name);
```

Parameters:

session_ch: Connection channel returned by wip_xxxClientCreate() / wip_xxxClientCreateOpts()
old_name: Previous name of the file
new_name: New name of the file

Returned Values:

0 is returned if the renaming file request has been sent successfully.
Negative error code is returned on failure.

2.10. Retrieving File Size

Internet Plug-In provide an API wip_getFileSize () to retrieve the file size in bytes. When this API successfully retrieves the file size, WIP_CEV_DONE event will be received in the event handler, otherwise WIP_CEV_ERROR event will be received.

Prototype:

```
s32 wip_getFileSize( wip_channel_t session_ch, ascii * file_name)
```

Parameters:

session_ch: Connection channel returned by wip_xxxClientCreate() / wip_xxxClientCreateOpts()
file_name: File name whose size needs to be retrieved

Returned Values:

0 is returned if the retrieving file size request has been sent successfully.
Negative error code is returned on failure.

2.11.Directory Listing

Internet Plug-In provide an API wip_list () for directory listing. Data channel will be created when this API is called.

Prototype:

wip_channel_t wip_list(wip_channel_t session_ch, ascii *dir_name, wip_eventHandler_f evh, void *ctx)

Parameters:

session_ch: Connection channel returned by wip_xxxClientCreate() / wip_xxxClientCreateOpts()
dir_name: Name of the directory whose content must be listed
evh: The event handler for the list channel.

Prototype:

void (*wip_eventHandler_f) (wip_channel_t Channel, wip_event_t Event, void *ctx)

Parameters of the call-back handler:

- Channel: Defines the list channel.
- Event: Structure indicating the event corresponding to the channel. It also carries the data related to the corresponding event. The various event and corresponding data that are notified are:
- **WIP_CEV_OPEN:** This event is sent when the list channel opens successfully.
- **WIP_CEV_READ:** This event signifies that the list data is being received from the server and can be read using the read API.
- **WIP_CEV_ERROR:** This event is sent in case of an error
- **WIP_CEV_PEER_CLOSE:** This event is sent when the last list data is read.
- ctx: Pointer to application context for a particular instance.

ctx: Pointer to application context

Returned Values:

List channel ID is returned on success.
Negative error code is returned on failure.

2.12. Initialization of wip_fileInfo_t Structure

Internet Plug-In provide an API wip_fileInfoInit () to initialize the wip_fileInfo_t structure.

Prototype:

wip_fileInfo_t *wip_fileInfoInit(void *buffer, u32 buf_len, ...)

Parameters:

buffer: Memory area where the file info structure will be built
buf_len: Buffer length
…: A list of entry descriptions terminated with WIP_FOPT_END

Returned Values:

Pointer to the wip_fileInfo_t structure is returned on success
NULL is returned on error.

Summary

The following points have been covered in this section

- **FILE APIs are used for creating the data channel which is used for data transfer when FTP/HTTP/SMTP/POP3 protocol are used.**
- **Following APIs are used for creating data channel**
  - **wip_getFile (creating a data channel with default parameters for downloading data)**
  - **wip_getFileOpts (creating a data channel with user defined parameters for downloading data)**
  - **wip_putFile (creating a data channel with default parameters for uploading data)**
  - **wip_putFileOpts (creating a data channel with user defined parameters for uploading data)**
  - **wip_list (creating list channel for retreiving the details of directory in list format)**
- **Following APIs are used for directory/file manipulation**
  - **wip_cwd (changing the current working directory)**
  - **wip_mkdir (creating new directory)**
  - **wip_deleteFile (deleting file)**
  - **wip_deleteDir (deleting empty directory)**
  - **wip_renameFile (renaming file)**
  - **wip_getFileSize (retreiving file size)**
  - **wip_fileInfoInit (initializing wip_fileInfo_t strcture)**

CHAPTER 41

FTP Client

1. Objective

This chapter will introduce you to FTP Client services provided by Internet Plug-In and will explain how to use them in your Open AT Application.

2. Introduction to FTP

File Transfer Protocol(FTP) is a TCP based service used for uploading/downloading a file from a FTP server. FTP uses two ports, a data port and a command port. Usually port 21 is used as command port and 20 is used as data port. The data port need not be 20 always it can be different port as well. There are two modes that FTP can operate:

- Active mode
- Passive mode

These two modes are initiated by the FTP client, and then acted upon by the FTP server.

In active mode, the client initiates a connection to the FTP server on its port 21 from a random unprivileged port (N > 1024). Port 21 is where the server is listening for commands issued to it, and in turn, which it will respond to. At this point the client begins to listen on it's command port + 1, and sends the PORT N+1 command to the server on its port 21. Once this is done the data transfer port (port 20) on the FTP server would initiate a connection to the FTP client's command port N+1 The main problem with FTP active mode actually falls on the client side. The FTP client doesn't make the actual connection to the data port of the server, it simply tells the server what port it is listening on and the server connects back to the specified port on the client. From the client side firewall this appears to be an outside system initiating a connection to an internal client, and hence it blocks. The way the FTP gets around this problem is by using passive FTP. The above scenario is depicted in the figure 16.

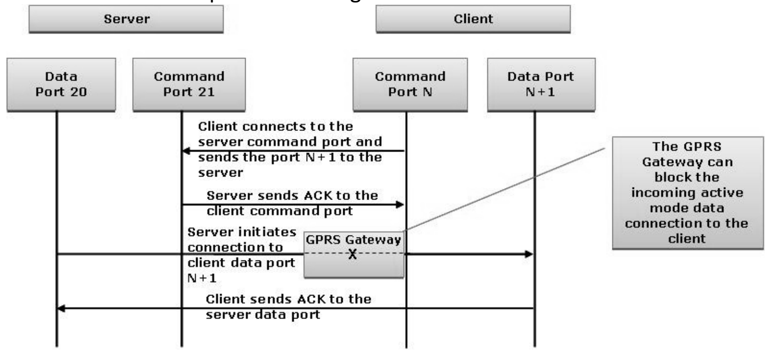

Server
Client
Data Port 20
Command Port 21
Command Port N
Data Port N+1
Client connects to the server command port and sends the port N+1 to the server
Server sends ACK to the client command port
Server initiates connection to client data port N+1
GPRS Gateway
X
Client sends ACK to the server data port
The GPRS Gateway can block the incoming active mode data connection to the client

Figure 104 – FTP Active mode illustration

In passive mode, the client starts up two connections to the FTP server i.e. it opens two random unprivileged ports locally (N > 1024 and N+1). The first port contacts the server on port 21, but instead of then issuing a PORT command and allowing the server to connect back to its data port (as in case of FTP active mode), the client will issue the FTP PASV command. The result of this is that the server then opens a random unprivileged port (P > 1024) and sends the PORT P command back to the client. The client then initiates the connection from port N+1 to port P on the server to transfer data.

In passive mode, the client initiates both connections to the server, solving the problem of firewalls filtering the incoming data port connection to the client from the server. The above scenario is depicted in the figure 17.

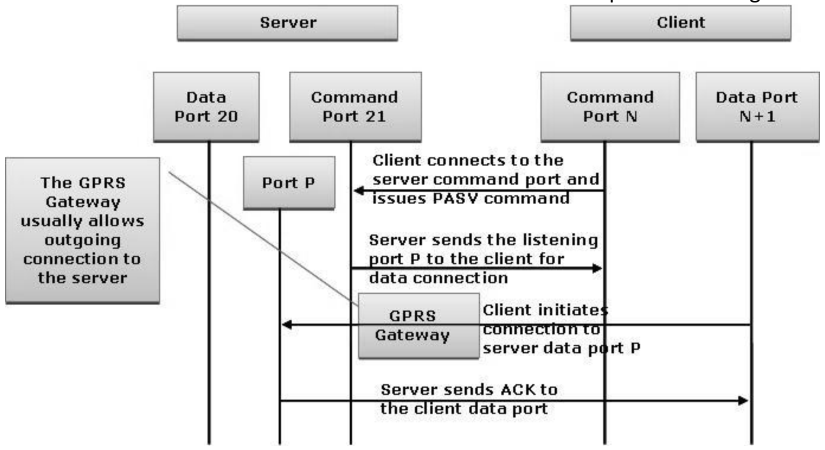

Server
Client
Data Port 20
Command Port 21
Command Port N
Data Port N+1
The GPRS Gateway usually allows outgoing connection to the server
Port P
Client connects to the server command port and issues PASV command
Server sends the listening port P to the client for data connection
GPRS Gateway
Client initiates connection to server data port P
Server sends ACK to the client data port

Figure 105– FTP Passive mode illustration

3. FTP APIs

The Internet Plug-In provide APIs for creating session/data channel with FTP protocol.
FTP requests are generated in two phases:
First, FTP session channel must be created using wip_FTPCreate () or wip_FTPCreateOpts (). This channel will store information such as FTP server address, user name, password etc.
A new data channel is then created using either wip_getFile () / wip_getFileOpts () or wip_putFile ()/ wip_putFileOpts () API for downloading or uploading file to FTP server.
In order to use this service the header file that need to be included is
wip_ftp.h

3.1. Creating of FTP Session Channel Without Options

Internet Plug-In provide an API wip_FTPCreate () to create the session channel for communicating with the FTP server. This API will create the FTP session channel with default parameters. Note that the FTP session channel will not be ready till WIP_CEV_OPEN event is received in the event handler.

Prototype:

```
wip_channel_t wip_FTPCreate( ascii *server_name, wip_eventHandler_f evh, void *ctx)
```

Parameters:

server_name: This indicates the FTP server name to which connection should be established. This can be either provided as DNS name or as a IP address.
evh: Event handler associated with the session channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the session channel is created successfully.
- WIP_CEV_ERROR: This event is sent in case of an error.
- WIP_CEV_PEER_CLOSE: This event is sent when the FTP server closes the connection.

Ctx: Pointer to application context.

Returned Values:

FTP session channel is returned on success.
NULL is returned on failure.

3.2. Creating of FTP Session Channel With Options

Internet Plug-In provide an API wip_FTPCreateOpts () to create the session channel for communicating with the FTP server. This API will create the FTP session channel with user defined parameters like mode of data transfer, user name, password etc. Note that the FTP session channel will not be ready till WIP_CEV_OPEN event is received in the event handler.

Prototype:

```
wip_channel_t wip_FTPCreateOpts( ascii *server_name, wip_eventHandler_f evh, void *ctx, ...);
```

Parameters:

server_name: This indicates the FTP server name to which connection should be established. This can be either provided as DNS name or as a IP address.
evh: Event handler associated with the session channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the session channel is created successfully.
- WIP_CEV_ERROR: This event is sent in case of an error.
- WIP_CEV_PEER_CLOSE: This event is sent when the FTP server closes the connection.

Ctx: Pointer to application context.
…: This provides the list of user defined options with values. The list should be terminated using WIP_COPT_END.

- WIP_COPT_TYPE: This indicates the data format used for transferring the data. By default "I" type is used for data transmission. This can be either one of the following values:
    - **I (Image):** In this mode, the data will be sent without any translation.
    - **B(Binary):** In this mode, the data will be sent in binary mode.
    - **A(ASCII):** In this mode, the data will be sent in Ascii mode.
- WIP_COPT_PASSIVE: This indicates the FTP operation mode. By default FTP is used in Passive mode. The can be either one of the following values:
    - **0 (Active):** The mode used for FTP operation is active.
    - **1 (Passive):** The mode used for FTP operation is passive.

WIP_COPT_USER: This indicates the login name for the FTP server. By default it is set to "anonymous".
WIP_COPT_PASSWORD: This indicates the password used for authentication. By default it is set to "wipftp@Sierra Wireless.com".

- WIP_COPT_ACCOUNT: This indicates the FTP account information. By default it is set to empty string.
- WIP_COPT_PEER_PORT: This indicates the port to be used for establishing connection with the FTP server. By default this will be set to 21.
- WIP_COPT_LIST_PLUGIN: This indicates the plug in handling the results from the LIST FTP command.
- WIP_COPT_KEEPALIVE: Sends the NOOP command for every nth second to keep the connection alive.
- WIP_COPT_FINALIZER: Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

Returned Values:

FTP session channel is returned on success.
NULL is returned on failure.

3.3. Configuring the FTP Session Channel

Internet Plug-In provide an API wip_setOpts() to configure a FTP session channel.

Prototype:

s32 wip_setOpts(wip_channel_t Channel, ...)

Parameters:

Channel: The FTP session channel to configure
...: List of options supported for this API.

- WIP_COPT_END: To mark the termination of the parameter list.
- WIP_COPT_KEEPALIVE: To send the NOOP command for every nth second to keep the connection alive
- WIP_COPT_SND_BUFSIZE: The size of the transmission buffer for client socket.
- WIP_COPT_RCV_BUFSIZE: The size of the receive buffer for the client socket.
- WIP_COPT_SND_LOWAT: Minimum amount of space that should be available in the transmission buffer. After the buffer is full, it triggers a WIP_CEV_WRITE event.
- WIP_COPT_RCV_LOWAT: This parameter signifies the minimum amount of space that should be available in the receive buffer before it triggers WIP_CEV_READ event.
- WIP_COPT_NODELAY: If this flag is set, then data is sent immediately.
- WIP_COPT_TOS: Type of service require, refer to RFC 791 for more details.
- WIP_COPT_TTL: The time for which the data is valid.
- WIP_COPT_TYPE: This indicates the data format used for transferring the data. By default "I" type is used for data transmission. This can be either one of the following values:
    - **I (Image):** In this mode, the data will be sent without any translation.
    - **B(Binary):** In this mode, the data will be sent in binary mode.
    - **A(ASCII):** In this mode, the data will be sent in Ascii mode.
- WIP_COPT_PASSIVE: This indicates the FTP operation mode. By default FTP is used in Passive mode. The can be either one of the following values:
    - **0 (Active): The mode used for FTP operation is active.**
    - **1 (Passive): The mode used for FTP operation is passive.**

- WIP_COPT_LIST_PLUGIN: This indicates the plug in handling the results from the LIST FTP command.
- WIP_COPT_FINALIZER: Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

Returned Values:

0 is returned on success.
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_NOT_SUPPORTED: This value is returned in case channel does not support this operation.
WIP_CERR_INVALID: This value is returned in case of an invalid option.

3.4. Retrieving the Configuration of a FTP Session Channel

Internet Plug-In provide an API wip_getOpts() to retrieve the configuration of a FTP Session channel.

Prototype:

```
s32 wip_getOpts(wip_channel_t Channel,…)
```

Parameters:

Channel: The channel from where the configuration should be retrieved.
…: List of options supported for this API.

- WIP_COPT_END: To mark the termination of the parameter list.
- WIP_COPT_SND_BUFSIZE: The size of the transmission buffer for client socket.
- WIP_COPT_RCV_BUFSIZE: The size of the receive buffer for the client socket.
- WIP_COPT_SND_LOWAT: Minimum amount of space that should be available in the transmission buffer. After the buffer is full, it triggers a WIP_CEV_WRITE event.
- WIP_COPT_RCV_LOWAT: This parameter signifies the minimum amount of space that should be available in the receive buffer before it triggers WIP_CEV_READ event.
- WIP_COPT_ERROR: This parameter indicates the error number corresponds to the last error encountered.
- WIP_COPT_NREAD: This parameter indicates the number of bytes that can be currently read on th socket.
- WIP_COPT_NWRITE: This parameter indicates the number of bytes that can be currently written on the socket.
- WIP_COPT_NODELAY: If this flag is set, then data is sent immediately.
- WIP_COPT_TOS: Type of service require, refer to RFC 791 for more details.
- WIP_COPT_TTL: The time for which the data is valid.
- WIP_COPT_PORT: Port occupied by the socket.
- WIP_COPT_STRADDR: Local address of the socket in string format.
- WIP_COPT_ADDR: Local address of the socket in IP address format.
- WIP_COPT_PEER_PORT: Port of the peer socket.
- WIP_COPT_PEER_STRADDR: Address of the peer socket in string format.
- WIP_COPT_PEER_ADDR: Address of the peer socket in IP address format.
- WIP_COPT_SUPPORT_READ: Fails if the channel does not support read operations. Otherwise it does nothing.
- WIP_COPT_SUPPORT_WRITE: Fails if the channel does not support write operations. Otherwise it does nothing.

- WIP_COPT_TYPE: This indicates the data format used for transferring the data. By default "I" type is used for data transmission. This can be either one of the following values:
    - **I (Image): In this mode, the data will be sent without any translation.**
    - **B(Binary): In this mode, the data will be sent in binary mode.**
    - **A(ASCII): In this mode, the data will be sent in Ascii mode.**
- WIP_COPT_PASSIVE: This indicates the FTP operation mode. By default FTP is used in Passive mode. The can be either one of the following values:
    - **0 (Active): The mode used for FTP operation is active.**
    - **1 (Passive): The mode used for FTP operation is passive.**
- WIP_COPT_LIST_PLUGIN: This indicates the plug in handling the results from the LIST FTP command.

Returned Values:

0 is returned on success.
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_NOT_SUPPORTED: This value is returned in case this channel does not support this operation.
WIP_CERR_INVALID: This value is retunred when invalid option is provided.
WIP_CERR_CSTATE: The channel is not ready for retrieving the configuration. This will be returned either when channel is still initializing or it is already closed.

3.5. Creating FTP Data Channel Without Options for Downloading File

Internet Plug-In provide an API wip_getFile () to create a data channel for downloading data from the FTP server. This API will create the data channel with default parameters.

Prototype:

wip_channel_t wip_getFile(wip_channel_t session_ch, ascii *file_name, wip_eventHandler_f evh, void *ctx)

Parameters:

session_ch: Connection channel returned by wip_FTPCreate() / wip_FTPCreateOpts()
file_name: Name of the file to be downloaded from the server
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully for exchanging data.
- WIP_CEV_READ: This event signifies that the data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event signifies that Internet Plug-In is ready to send data.
- WIP_CEV_ERROR: This event is sent in case of an error
- WIP_CEV_PEER_CLOSE: This event is sent when the complete data is downloaded.

Ctx: Pointer to application context.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

3.6. Creating Data Channel With Options for Downloading File

Internet Plug-In provide an API wip_getFileOpts () to create a data channel for downloading data from the FTP server. This API will create the data channel with the options specific to protocol.

Prototype:

wip_channel_t wip_getFileOpts(wip_channel_t session_ch, ascii *file_name, wip_eventHandler_f evh, void *ctx, ...)

Parameters:

session_ch: Connection channel returned by wip_FTPCreate() / wip_FTPCreateOpts()
file_name: Name of the file to be downloaded from the server
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully for exchanging data.
- WIP_CEV_READ: This event signifies that the data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event signifies that Internet Plug-In is ready to send data.
- WIP_CEV_ERROR: This event is sent in case of an error
- WIP_CEV_PEER_CLOSE: This event is sent when the complete data is downloaded.
- Ctx: Pointer to application context.
- ...: This specifies the option specific to protocol with the option value. Note that the list must be terminated by WIP_COPT_END.
- WIP_COPT_FILE_NAME: Name of the file to be downloaded.
- WIP_COPT_RESTART: Restart the download after the nth byte.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

3.7. Creating Data Channel Without Options for Uploading File

Internet Plug-In provide an API wip_putFile () to create a data channel for uploading data to the FTP server. This API will create the data channel with default parameters.

Prototype:

wip_channel_t wip_putFile(wip_channel_t session_ch, ascii *file_name, wip_eventHandler_f evh, void *ctx)

Parameters:

session_ch: Connection channel returned by wip_FTPCreate() / wip_FTPCreateOpts()
file_name: Name of the file to be uploaded to the server
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully for exchanging data.
- WIP_CEV_READ: This event signifies that the data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event signifies that Internet Plug-In is ready to send data.
- WIP_CEV_ERROR: This event is sent in case of an error
- WIP_CEV_PEER_CLOSE: This event is sent when the complete data is downloaded.

Ctx: Pointer to application context.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

3.8. Creating Data Channel With Options for Uploading File

Internet Plug-In provide an API wip_putFileOpts () to create a data channel for uploading data to the FTP server. This API will create the data channel with the options specific to protocol.

Prototype:

wip_channel_t wip_putFileOpts(wip_channel_t session_ch, ascii *file_name, wip_eventHandler_f evh, void *ctx, ...)

Parameters:

session_ch: Connection channel returned by wip_FTPCreate() / wip_FTPCreateOpts()
file_name: Name of the file to be uploaded to the server
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully for exchanging data.
- WIP_CEV_READ: This event signifies that the data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event signifies that Internet Plug-In is ready to send data.
- WIP_CEV_ERROR: This event is sent in case of an error
- WIP_CEV_PEER_CLOSE: This event is sent when the complete data is downloaded.

Ctx: Pointer to application context.
...: This specifies the option specific to protocol with the option value. Note that the list must be terminated by WIP_COPT_END.

- WIP_COPT_FILE_NAME: Name of the file to be uploaded.
- WIP_COPT_RESTART: Restart the upload after the nth byte.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

3.9. Reading Data From FTP Data Channel

Internet Plug-In provide an API wip_read() which is used to read the file data. Note that this function is not supported by FTP session channel.

Prototype:

```
s32 wip_read (wip_channel_t Channel, void *buffer, u32 buf_len)
```

Parameters:

Channel: FTP data channel returned by either wip_getFile () / wip_getFileOpts () / wip_putFile () / wip_putFileOpts ()
buffer: The buffer where data will be copied.
buf_len: Length of the buffer.

Returned Values:

The number of bytes read from the channel.
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_CSTATE: The channel is not ready to receive data.
WIP_CERR_NOT_SUPPORTED: This channel does not support data read operation.

3.10. Writing Data To The FTP Data Channel

Internet Plug-In provide an API wip_write() to write the file data. Note that this function is not supported by FTP session channel.

Prototype:

```
s32 wip_write (wip_channel_t Channel, void *buffer, u32 buf_len)
```

Parameters:

Channel: FTP data channel returned by either wip_getFile () / wip_getFileOpts () / wip_putFile () / wip_putFileOpts ().
buffer: The buffer to write.
buf_len: Length of the buffer.

Returned Values:

The number of bytes written to the channel.
In case of an error the various negative error codes that are returned are described below:
WIP_CERR_CSTATE: The channel is not ready to accept data.
WIP_CERR_NOT_SUPPORTED: This channel does not support data writing operation.

3.11. Shutting FTP Data Channel

Internet Plug-In provide an API wip_shutdown() to shut either the input or output communication of a given socket. If both communications are shut down, the socket is closed. This API is used to indicate the end of file while uploading the file to the FTP server.

Prototype:

s32 wip_shutdown (wip_channel_t Channel, bool read, bool write)

Parameters:

Channel: The channel ID of the socket to be shutdown.
read: If this parameter is set to TRUE, input communication over the socket will closed down. In other words this socket will not be able to receive any data.
write: If this parameter is set to TRUE, then output communication will be closed down for the given channel. In other words this socket will not be able to send any data.

Returned Values:

0 is returned on success.
In case of an error a negative value is returned.
WIP_CERR_NOT_SUPPORTED: This value is returned in case this API is used to shutdown either TCP server or UDP channel.
WIP_CERR_INTERNAL: This value is returned when there is a internal problems to shutdown the channel.

3.12. Closing FTP Session/Data Channel

Internet Plug-In provide an API wip_close() to close the FTP session or data channel. When this API is used to close FTP session channel, it releases all the resources associated with the session channel. Note that closing session channel will not close the data channel. The data channel should be closed explicitly by calling this function. Closing the data channel will make the session ready for another request.

Prototype:

s32 wip_close(wip_channel_t Channel)

Parameters:

Channel: FTP session/data channel that should be closed.

Returned Values:

0 is returned on success.
In case of an error, it returns the following:
WIP_CERR_MEMORY: Insufficient memory
WIP_CERR_INVALID: Specified NULL channel

4. Sample Code

```
#include "adl_global.h"
#include "wip_net.h"
#include "wip_bearer.h"
#include "wip_ftp.h"

const u16 wm_apmCustomStackSize = 4096;

#define GPRS_APN      "airtelgprs.com"
#define GPRS_LOGIN    "airtel"
#define GPRS_PASSWORD "tango"

#define FTP_PORT    21
#define FTP_MODE   1
const ascii * FTP_TYPE = 'A';
const ascii * FTP_STR_HOSTNAME   = "81.80.89.163";
const ascii * FTP_STR_USERNAME   = "ndt-ftp";
const ascii * FTP_STR_PASSWORD   = "ndt-ftp";
const ascii * FTP_FILENAME = "./test.txt";

wip_bearer_t bearerHandle = NULL;
wip_channel_t ftpCnxCh = NULL;
wip_channel_t ftpDataCh = NULL;

void simHandler ( u8 Event );
void evh_bearer( wip_bearer_t b, s8 event, void *ctx);
void FTPCnxHandler(wip_event_t *ev, void *ctx );
void ftpDataHandler(wip_event_t *ev, void *ctx );

void ftpDataHandler(wip_event_t *ev, void *ctx )
{
   switch( ev->kind)
   {
    case WIP_CEV_OPEN:
      TRACE((1,"ftpDataHandler: WIP_CEV_OPEN"));
      break;
    case WIP_CEV_READ:
      {
        u8 buf[256 ];
        wm_memset(buf,0,256);

        TRACE((1,"ftpDataHandler: WIP_CEV_READ"));
        while( wip_read(ftpDataCh, buf, 256 ) > 0 )
        {
            TRACE((1, buf));
        }
      }
      break;
    case WIP_CEV_WRITE:
      TRACE((1,"ftpDataHandler: WIP_CEV_WRITE"));
      break;
    case WIP_CEV_PEER_CLOSE:
      TRACE((1,"ftpDataHandler: WIP_CEV_PEER_CLOSE"));
      wip_close(ftpDataCh);
      wip_close(ftpCnxCh);
      break;
    case WIP_CEV_ERROR:
      TRACE((1,"ftpDataHandler: WIP_CEV_ERROR"));
      break;
  }
}

void FTPCnxHandler(wip_event_t *ev, void *ctx )
{
  switch( ev->kind)
  {
    case WIP_CEV_OPEN:
      TRACE((1,"WIP_CEV_OPEN"));
      ftpDataCh =  wip_getFileOpts(ftpCnxCh,
                              FTP_FILENAME,
                              ftpDataHandler,
                              NULL,
                              WIP_COPT_END);
     TRACE (( 1, "wip_getFileOpts: %x",ftpDataCh ));
      break;
    case WIP_CEV_PEER_CLOSE:
      TRACE((1,"WIP_CEV_PEER_CLOSE"));
      break;
    case WIP_CEV_ERROR:
      TRACE((1,"WIP_CEV_ERROR"));
      break;
  }
}

void evh_bearer( wip_bearer_t b, s8 event, void *ctx)
{
  switch(event)
  {
    case WIP_BEV_IP_CONNECTED:
      TRACE (( 1, "evh_bearer: <WIP_BEV_IP_CONNECTED>"));
      /* Start the FTP session */
      ftpCnxCh =  wip_FTPCreateOpts( FTP_STR_HOSTNAME,
                                FTPCnxHandler ,
                                NULL,
                                WIP_COPT_USER, FTP_STR_USERNAME,
                    WIP_COPT_PASSWORD, FTP_STR_PASSWORD,
                    WIP_COPT_PASSIVE, FTP_MODE,
                    WIP_COPT_TYPE, FTP_TYPE,
                    WIP_COPT_PEER_PORT, FTP_PORT,
                    WIP_COPT_END);
   TRACE (( 1, "wip_FTPCreateOpts: %x",ftpCnxCh ));
   break;
  case WIP_BEV_CONN_FAILED:
   TRACE (( 1, "evh_bearer: <WIP_BEV_CONN_FAILED>"));
   wip_bearerClose( bearerHandle );
   break;
  case WIP_BEV_STOPPED:
   TRACE (( 1, "evh_bearer: <WIP_BEV_STOPPED>"));
   wip_bearerClose(bearerHandle );
   break;
  default:
   TRACE (( 1, "evh_bearer: <Other Events>"));
   break;
  }
}

void simHandler ( u8 Event )
{
 if( ADL_SIM_EVENT_FULL_INIT == Event )
 {
  s8 ret;
  /* Open the GPRS bearer */
  ret = wip_bearerOpen ( &bearerHandle, "GPRS", ( wip_bearerHandler_f ) evh_bearer, NULL);
  TRACE (( 1, "wip_bearerOpen: ret = %d",ret ));
  ret = wip_bearerSetOpts( bearerHandle,
               WIP_BOPT_GPRS_APN, GPRS_APN,
               WIP_BOPT_LOGIN,    GPRS_LOGIN,
               WIP_BOPT_PASSWORD, GPRS_PASSWORD,
               WIP_BOPT_END);
  TRACE (( 1, "wip_bearerSetOpts: ret = %d",ret ));
  ret = wip_bearerStart( bearerHandle );
 }
}

void adl_main (adl_InitType_e adlInitType)
{
 TRACE((1,"Main function"));
 /* Subscribe to the SIM events */
 adl_simSubscribe (( adl_simHdlr_f ) simHandler, NULL);

 /* Initialize the stack */
 wip_netInit();
}
```

Summary

The following points have been covered in this section

- **FTP APIs are used for creating FTP session and data channel.**
- **Following APIs are used for creating FTP session channel**
    - **wip_FTPCreate (creating a FTP session channel with default parameter)**
    - **wip_FTPCreateOpts (creating a FTP session channel with userd defined parameters)**
- **Following APIs are used for creating FTP data channel**
    - **wip_getFile (creating a data channel with default pa**
    - **rameters for downloading file)**
    - **wip_getFileOpts (creating a data channel with user defined parameters for downloading file)**
    - **wip_putFile (creating a data channel with default parameters for uploading file)**
    - **wip_putFileOpts (creating a data channel with user defined parameters for uploading file)**
- **wip_getOpts API is used for retrieving the configuration parameters from the FTP session channel**
- **wip_setOpts API is used for configuring the parameters of the FTP session channel**
- **wip_read API is used for reading file data from the FTP data channel**
- **wip_write API is used for writing file data to the FTP data channel**
- **wip_shutdown API is used for shutting down the FTP data channel**
- **wip_close API is used for closing both FTP session and data channel**

CHAPTER 42

HTTP Client

1. Objective

This chapter will introduce you to HTTP Client services provided by Internet Plug-In and will explain how to use them in your Open AT Application.

2. Introduction to HTTP

HTTP is an application level protocol of the TCP/ IP suite, which is used to deliver virtually all files and other data on the World Wide Web. It is used to transmit resources that are identified by a URL. The most common kind of a resource can be a file, but it can also be dynamically generated content, which is the result of execution of a script or an application on the server.

HTTP is a request / response based protocol operating between Web Clients and Web Servers. A browser is also known as a HTTP Client because it sends requests to a HTTP server. HTTP servers listen and respond to incoming requests.

3. HTTP Request-Response Model

HTTP is a simple Request-Response protocol. A HTTP client, such as a Web Browser initiates a request by establishing a TCP/ IP connection to a particular port on a remote host (port: 80 by default). A HTTP server listening on that port waits for the client to send a request. Upon receiving the request, the server sends back a response.

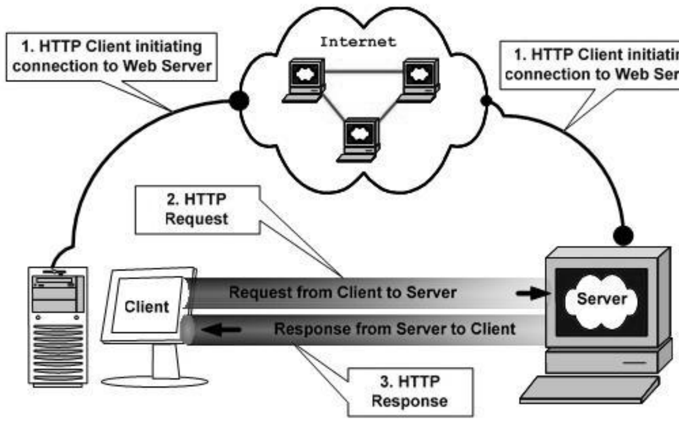

1. HTTP Client initiating connection to Web Server
Internet
1. HTTP Client initiating connection to Web Server
2. HTTP Request
Client
Request from Client to Server
Response from Server to Client
Server
3. HTTP Response

Figure 106 - HTTP Request-Response Model

Thus, the Client sends a request to the server (HTTP Request). The server processesthe request and sends a response to the client (HTTP Response).

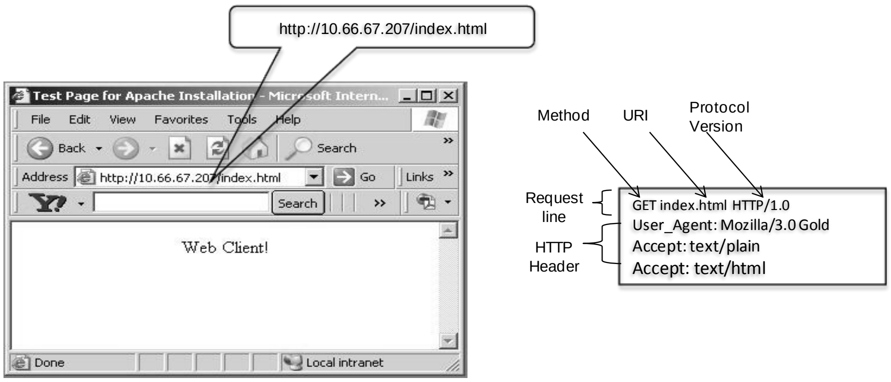
4. HTTP Request

The HTTP request has the following message format for transferring entities:

- A Request line
- Zero or more header lines
- A Blank line which separates the headers from the message body
- Message body
- The Request line of the HTTP request (from a client to a server) includes:
- The method to be applied on the resource
- The identifier of the resource
- The HTTP protocol version in use

The Method field in request line of HTTP indicates the method to be performed on the object identified by the URL. Some of the methods are listed below:

- GET
- HEAD
- POST
- PUT
- DELETE


http://10.66.67.207/index.html
Test Page for Apache Installation - Microsoft Intern...
File Edit View Favorites Tools Help
Address http://10.66.67.207/index.html
Web Client!
Done
Local intranet
Method
URI
Protocol Version
Request line
HTTP Header

```
GET index.html HTTP/1.0
User_Agent: Mozilla/3.0 Gold
Accept: text/plain
Accept: text/html
```

Figure 107 - HTTP Request Message

The second part of the request line is the Uniform Resource Identifier (URI) that identifies the resource upon which the request has to be applied. The last part of the request line specifies the version of HTTP.
The request also contains optional header lines. It provides additional information about the request to the server. For example,
User_Agent is a header that indicates the type of client being used.
User_Accept indicates to the server, the type of data that the client can accept
In the HTTP request message, the message body is optional and it is influenced by the choice of the method.

5. HTTP Response

In response to a HTTP Request sent by a HTTP Client (Web Browser typically), the server sends a HTTP Response. The HTTP response to requests is usually a program output (Dynamic content) and not a static file.
The first line of a Response message is the Status-Line. It consists of:

- The protocol version
- Numeric status code
- Description of the status code

HTTP Status Codes: The response status line contains the status of processing of the HTTP Request. In case of success, it will contain the status code 200 and description is OK. The status line in this case will be “HTTP/ 1. 0 200 OK“.
In case of Error, the server sends an appropriate error code back to the client . The HTTP error codes are standardized. Some of the commonly found error codes:

- HTTP/ 1. 0 404 Not Found
- HTTP/ 1. 0 500 Internal Server Error

The Error/Success codes of the HTTP response are standardized in the following manner:

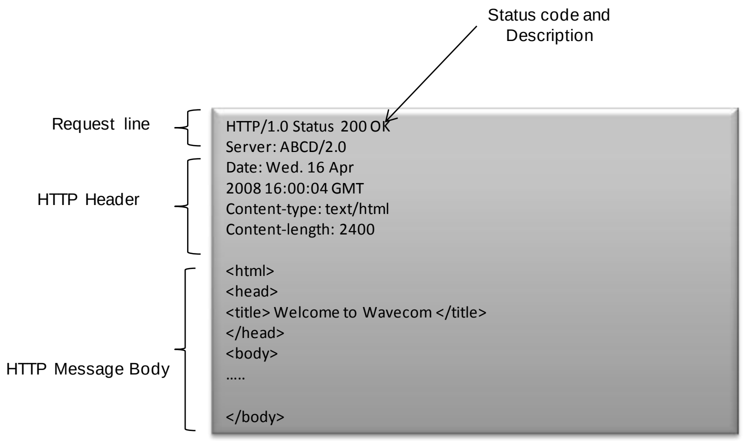
- 1xx: indicates informational message only
- 2xx: indicates success of some kind
- 3xx: redirects the client to another URL
- 4xx: indicates an error on the client' s part
- 5xx: indicates an error on the server' s part

6. HTTP Response Headers

The response must contain Header lines describing the following:

- MIME-type19 of the data being sent in response
- Date and Time stamp
- Content Size etc.
-

The HTTP Response message body contains the requested data.



```
HTTP/1.0 Status 200 OK
Server: ABCD/2.0
Date: Wed. 16 Apr
2008 16:00:04 GMT
Content-type: text/html
Content-length: 2400

<html>
<head>
<title> Welcome to Wavecom </title>
</head>
<body>
.....

</body>
```

Figure 108 - HTTP Response Message

7. HTTP Methods

HTTP method specifies the approach that is to be applied on a resource identified by URL. HTTP has many methods. The most commonly used ones are GET, POST and HEAD methods.

GET Method: The GET method is the most frequently used method. It is used by default to get static content. This method can also be used to submit data from a HTML web page to the server. In GET method, the data submitted will be sent as part of the URL. Hence, in GET method parameters are encoded and passed along with the URL. Usually, parameters are passed as name-value pairs. As GET method sends data as part of URL, there is a physical restriction on the size of it being sent.

POST Method: A POST method is used to send data as part of the HTTP message body. In certain cases, the client may need to send megabytes of information. In these situations, POST method is the right choice.
A POST request passes all its data of unlimited length, directly as part of its HTTP request body. The exchange is invisible to the client. The URL does not contain the data submitted. Consequently, POST requests cannot be book marked or emailed or in some cases, even reloaded. Hence, confidential information sent to the server, such as the credit card number, should be sent via POST method. Moreover, because GET requests can be easily book marked, they should not be used in situations like placing an order or updating a database.
Note: Never use GET method when passing sensitive data like passwords, credit card numbers, etc.

HEAD Method: The HEAD method is similar to the GET method, except that it asks the server to return only the Response headers and not the content (i.e., No Message Body). This method is useful for the clients to check the characteristics of the resource without actually downloading it, thus saving bandwidth. The HTTP Clients usually use the HEAD method when they do not need the file' s contents.
HEAD method is used for example:

- To determine the document's size
- To know document's modification time
- To know general availability of a web page

PUT Method: The PUT method can be used to put a page at a specific URL. If there's already a page there, it deletes the existing page and creates the new page. If there's no page there, a new one is created. The PUT method is a much more limited operation that never does anything more than PUT one page at a specified URL.

Both PUT and POST can be used to create new pages. However PUT should be used when the client specifies the location for the page. PUT is normally the right protocol for a web editor like DreamWeaver or BBEdit. POST is used when the client sends the page to the server, and the server then tells the client where to put the page. POST is normally the right protocol for a blog editor like TypePad or anything that inputs into a content management system.
DELETE Method: The DELETE method is used to delete the resource identified by the Request URI. Note that all the servers don't support this method as this method can be used by the client to delete the resource at the server.

NOTE :
DELETE method is optional and some servers may not support the same.

8. HTTP Versions

HTTP 1.0 was the original version. In HTTP 1.0, the client would initiate and close the connection to the server on each request. For example, if a page index.html is accessed by the client and the page contains ten image files in its body, the browser would have to initiate and close the connection eleven times. i.e., the client opens a new TCP/ IP connection and places a request for the resource (first request). On receipt of the response, the connection is closed. Further, similar request - reply transaction (ten transactions) happens for each picture to be downloaded.

Performance problems with HTTP1.0: HTTP 1.0 suffers from poor performance because it makes new TCP/ IP connection for every request . This significantly increases the overhead associated with opening those connections. Further, the client is not able to send requests until the reply to the previous one has been received. The performance becomes worse when the network is busy.

The latest version of HTTP is HTTP 1.1. In this version of HTTP, the connection established is maintained until the browser is closed. This feature is known as persistent connection. Persistent connections allow many requests to occur within a single TCP/ IP connection. i.e. , instead of opening multiple TCP connections for each object to be retrieved, one TCP/ IP connection is opened and multiple HTTP requests are issued through that connection.

9. HTTP APIs

The Internet Plug-In provide APIs for generating HTTP requests. The Sierra Wireless Internet Plug-In provide following features:

- Support for HTTP version 1.1 (default) and 1.0
- Persistent connections (with HTTP 1.1)
- Connection to a HTTP proxy server
- Basic and digest (MD5) authentication
- Chunked transfer coding
- Setting HTTP request headers
- Getting HTTP response headers
- GET, HEADER, POST and PUT methods

HTTP requests are generated in two phases:

First, HTTP request channel must be created using wip_HTTPClientCreate () or wip_HTTPClientCreateOpts (). This channel will store information such as HTTP version, address of proxy server, HTTP request header which are common to all further HTTP requests. Note that, when HTTP 1.1 is used, a new TCP channel is not created for each request is sent to the same server or proxy, instead the TCP connection is maintained by the HTTP session whenever possible.

A new data channel is then created using either wip_getFile () / wip_getFileOpts () or wip_putFile ()/ wip_putFileOpts () API for each HTTP GET/HEADER/POST/PUT request.

In order to use this service the header file that need to be included is
wip_http.h

9.1. Creating of HTTP Session Channel Without Options

Internet Plug-In provide an API wip_HTTPClientCreate () to create the HTTP session channel. This API will create the HTTP session channel with default parameters.

Prototype:

```
wip_channel_t wip_HTTPClientCreate(wip_eventHandler_f evh, void *ctx)
```

Parameters:

evh: Event handler associated with the session channel. Currently, there is no event defined for HTTP session and hence it can be set to NULL.
Ctx: Pointer to application context.

Returned Values:

HTTP session channel is returned on success.
NULL is returned on failure.

9.2. Creating of HTTP Session Channel With Options

Internet Plug-In provide an API wip_HTTPClientCreateOpts () to create the HTTP session channel. This API will create the HTTP session channel with default parameters.

Prototype:

```
wip_channel_t wip_HTTPClientCreateOpts( wip_eventHandler_f evh, void *ctx, ...)
```

Parameters:

evh: Event handler associated with the session channel. Currently, there is no event defined for HTTP session and hence it can be set to NULL.
Ctx: Pointer to application context.

- …: This provides the list of user defined options with values. The list should be terminated using WIP_COPT_END.
- WIP_COPT_END: To mark the termination of the parameter list.
- WIP_COPT_RCV_BUFSIZE: Sets the size of the TCP socket receive buffer.
- WIP_COPT_SND_BUFSIZE: Sets the size of the TCP socket send buffer.

- WIP_COPT_PROXY_STRADDR: Sets the hostname of the HTTP proxy server, a NULL value disables the proxy server.
- WIP_COPT_PROXY_PORT: Sets the port number of the HTTP proxy server, the default value is 80.
- WIP_COPT_HTTP_VERSION: Defines the HTTP version to be used by the session.
- WIP_COPT_HTTP_HEADER: This option adds a HTTP message header field that will be sent on each request. The first value is the field name (without the colon), the second value is the field value (without CRLF), a NULL value can be passed to remove a previously defined header field.
- WIP_COPT_HTTP_HEADER_LIST: This option adds a list of HTTP message header fields to send with each request. The value points to an array of wip_httpHeader_t structures, the last element of the array has its name field set to NULL.
- WIP_COPT_FINALIZER: Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

Returned Values:

HTTP session channel, a positive integer, is returned on success.
NULL is returned on failure.

9.3. Creating HTTP Data Channel Without Options for Downloading URL Data

Internet Plug-In provide an API wip_getFile () to create a data channel for downloading data from the specified URL. This API will create the data channel with default parameters. When this API is used, HTTP GET method is used to send request.

Prototype:

wip_channel_t wip_getFile(wip_channel_t session_ch, ascii *url, wip_eventHandler_f evh, void *ctx)

Parameters:

session_ch: Session channel returned by wip_HTTPClientCreate() / wip_HTTPClientCreateOpts()
url: Name of the URL to which HTTP request has to be sent.
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully. This event is received when response message header has been received. The wip_getOpts () API can be used to retrieve the header information:
- WIP_COPT_HTTP_STATUS_CODE: 3-digit response status code
- WIP_COPT_HTTP_STATUS_REASON: the reason phrase
- WIP_COPT_HTTP_HEADER: value of response header fields.
- WIP_CEV_READ: This event signifies that the response message body data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event is sent when request message body data can be written by the application using write API.
- WIP_CEV_ERROR: This event is sent in case of an error.
- WIP_CEV_PEER_CLOSE: This event is sent after the entire response message, including response header and response body data, has been received.

Ctx: Pointer to application context.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

9.4. Creating HTTP Data Channel With Options for Downloading URL Data

Internet Plug-In provide an API wip_getFileOpts () to create a data channel for downloading data from the specified URL. This API will create the data channel with user defined parameters.

Prototype:

wip_channel_t wip_getFileOpts(wip_channel_t session_ch, ascii *url, wip_eventHandler_f evh, void *ctx, …)

Parameters:

session_ch: Session channel returned by wip_HTTPClientCreate() / wip_HTTPClientCreateOpts()
url: Name of the URL to which HTTP request has to be sent.
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully. This event is received when response message header has been received. The wip_getOpts () API can be used to retrieve the header information:
- WIP_COPT_HTTP_STATUS_CODE: 3-digit response status code
- WIP_COPT_HTTP_STATUS_REASON: the reason phrase
- WIP_COPT_HTTP_HEADER: value of response header fields.
- WIP_CEV_READ: This event signifies that the response message body data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event is sent when request message body data can be written by the application using write API.
- WIP_CEV_ERROR: This event is sent in case of an error.
- WIP_CEV_PEER_CLOSE: This event is sent after the entire response message, including response header and response body data, has been received.
- Ctx: Pointer to application context.
- …: This specifies the option specific to protocol with the option value. Note that the list must be terminated by WIP_COPT_END.
- WIP_COPT_END: To mark the termination of the parameter list.
- WIP_COPT_HTTP_METHOD: Defines the HTTP method to be used by the session. By default, HTTP GET method is used for the session. Other supported methods are HTTP HEADER, HTTP POST and HTTP PUT method.
- WIP_COPT_HTTP_HEADER: This option adds a HTTP message header field that will be sent on each request. The first value is the field name (without the colon), the second value is the field value (without CRLF), a NULL value can be passed to remove a previously defined header field.
- WIP_COPT_HTTP_HEADER_LIST: This option adds a list of HTTP message header fields to send with each request. The value points to an array of wip_httpHeader_t structures, the last element of the array has its name field set to NULL.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

9.5. Creating HTTP Data Channel Without Options for Uploading URL Data

Internet Plug-In provide an API wip_putFile () to create a data channel for uploading data to the specified URL. This API will create the data channel with default parameters. When this API is used, HTTP PUT method is used to send request.

Prototype:

wip_channel_t wip_putFile(wip_channel_t session_ch, ascii *url, wip_eventHandler_f evh, void *ctx)

Parameters:

session_ch: Session channel returned by wip_HTTPClientCreate() / wip_HTTPClientCreateOpts()
url: Name of the URL to which HTTP request has to be sent.
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully. This event is received when response message header has been received. The wip_getOpts () API can be used to retrieve the header information:
- WIP_COPT_HTTP_STATUS_CODE: 3-digit response status code
- WIP_COPT_HTTP_STATUS_REASON: the reason phrase
- WIP_COPT_HTTP_HEADER: value of response header fields.
- WIP_CEV_READ: This event signifies that the response message body data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event is sent when request message body data can be written by the application using write API.
- WIP_CEV_ERROR: This event is sent in case of an error.
- WIP_CEV_PEER_CLOSE: This event is sent after the entire response message, including response header and response body data, has been received.

Ctx: Pointer to application context.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

9.6. Creating HTTP Data Channel With Options for Uploading URL Data

Internet Plug-In provide an API wip_putFileOpts () to create a data channel for uploading data to the specified URL. This API will create the data channel with user defined parameters.

Prototype:

wip_channel_t wip_putFileOpts(wip_channel_t session_ch, ascii *url, wip_eventHandler_f evh, void *ctx, …)

Parameters:

session_ch: Session channel returned by wip_HTTPClientCreate() / wip_HTTPClientCreateOpts()
url: Name of the URL to which HTTP request has to be sent.
evh: Event handler associated with the data channel. This handler receive following events:

- WIP_CEV_OPEN: This event signifies that the data channel is created successfully. This event is received when response message header has been received. The wip_getOpts () API can be used to retrieve the header information:
- WIP_COPT_HTTP_STATUS_CODE: 3-digit response status code
- WIP_COPT_HTTP_STATUS_REASON: the reason phrase
- WIP_COPT_HTTP_HEADER: value of response header fields.
- WIP_CEV_READ: This event signifies that the response message body data is being received from the server and can be read using the read API.
- WIP_CEV_WRITE: This event is sent when request message body data can be written by the application using write API.
- WIP_CEV_ERROR: This event is sent in case of an error.
- WIP_CEV_PEER_CLOSE: This event is sent after the entire response message, including response header and response body data, has been received.
- Ctx: Pointer to application context.
- …: This specifies the option specific to protocol with the option value. Note that the list must be terminated by WIP_COPT_END.
- WIP_COPT_END: To mark the termination of the parameter list.
- WIP_COPT_HTTP_METHOD: Defines the HTTP method to be used by the session. By default, HTTP GET method is used for the session. Other supported methods are HTTP HEADER, HTTP POST and HTTP PUT method.
- WIP_COPT_HTTP_HEADER: This option adds a HTTP message header field that will be sent on each request. The first value is the field name (without the colon), the second value is the field value (without CRLF), a NULL value can be passed to remove a previously defined header field.
- WIP_COPT_HTTP_HEADER_LIST: This option adds a list of HTTP message header fields to send with each request. The value points to an array of wip_httpHeader_t structures, the last element of the array has its name field set to NULL.

Returned Values:

Created data channel is returned on success.
NULL is returned on failure.

9.7. Reading Data From HTTP Data Channel

Internet Plug-In provide an API wip_read () which is used to read the response message body. Note that this function is not supported by HTTP session channel.

Prototype:

s32 wip_read (wip_channel_t Channel, void *buffer, u32 buf_len)

Parameters:

Channel: HTTP request channel returned by either wip_getFile () / wip_getFileOpts () / wip_putFile () / wip_putFileOpts ()
buffer: The buffer where data will be copied.
buf_len: Length of the buffer.

Returned Values:

The number of bytes read from the channel is returned on success.
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_CSTATE: The channel is not ready to receive data
WIP_CERR_NOT_SUPPORTED: This channel does not support data read operation

9.8. Writing Data To The HTTP Data Channel

Internet Plug-In provide an API wip_write() to write the request message body. Note that this function is not supported by HTTP session channel.

Prototype:

s32 wip_write (wip_channel_t Channel, void *buffer, u32 buf_len)

Parameters:

Channel: HTTP request channel returned by either wip_getFile () / wip_getFileOpts () / wip_putFile () / wip_putFileOpts ().
buffer: The buffer to write.
buf_len: Length of the buffer.

Returned Values:

The number of bytes written to the channel is returned on success.
In case of an error the various negative error codes that are returned are described below:
WIP_CERR_CSTATE: The channel is not ready to accept data.
WIP_CERR_NOT_SUPPORTED: This channel does not support data writing operation.

9.9. Shutting HTTP Data Channel

Internet Plug-In provide an API wip_shutdown() to shut either the input or output communication of a given socket. If both communications are shut down, the socket is closed. on a request channel to signals the end of the message body, it has no effect if the request has no message body. Note that this function is not supported by HTTP session channel.

Prototype:

s32 wip_shutdown (wip_channel_t Channel, bool read, bool write)

Parameters:

Channel: HTTP request channel returned by either wip_getFile () / wip_getFileOpts () / wip_putFile () / wip_putFileOpts ().
read: If this parameter is set to TRUE, input communication over the socket will closed down. In other words this socket will not be able to receive any data.
write: If this parameter is set to TRUE, then output communication will be closed down for the given channel. In other words this socket will not be able to send any data.

Returned Values:

0 is returned on success.
In case of an error a negative value is returned.
WIP_CERR_NOT_SUPPORTED: This value is returned in case this API is used to shutdown either TCP server or UDP channel.
WIP_CERR_INTERNAL: This value is returned when there is a internal problems to shutdown the channel.

9.10. Configuring the HTTP Session Channel

Internet Plug-In provide an API wip_setOpts() to configure a HTTP session channel. Note that this function is not supported by HTTP data channel.

Prototype:

s32 wip_setOpts(wip_channel_t Channel, ...)

Parameters:

Channel: HTTP session channel to configure.
...: List of options supported for this API.

- WIP_COPT_END: To mark the termination of the parameter list.
- WIP_COPT_RCV_BUFSIZE: Sets the size of the TCP socket receive buffer.
- WIP_COPT_SND_BUFSIZE: Sets the size of the TCP socket send buffer.
- WIP_COPT_PROXY_STRADDR: Sets the hostname of the HTTP proxy server, a NULL value disables the proxy server.
- WIP_COPT_PROXY_PORT: Sets the port number of the HTTP proxy server, the default value is 80.
- WIP_COPT_HTTP_VERSION: Defines the HTTP version to be used by the session.
- WIP_COPT_HTTP_HEADER: This option adds a HTTP message header field that will be sent on each request. The first value is the field name (without the colon), the second value is the field value (without CRLF), a NULL value can be passed to remove a previously defined header field.
- WIP_COPT_HTTP_HEADER_LIST: This option adds a list of HTTP message header fields to send with each request. The value points to an array of wip_httpHeader_t structures, the last element of the array has its name field set to NULL.

- WIP_COPT_FINALIZER: Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

Returned Values:
0 is returned on success.
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_NOT_SUPPORTED: This channel does not support this operation.
WIP_CERR_INVALID: Invalid option.
Retrieving the Configuration of a HTTP Session/Data Channel
Internet Plug-In provide an API wip_getOpts() to retrieve the configuration of a HTTP Session channel.

Prototype:

```
s32 wip_getOpts(wip_channel_t Channel,…)
```

Parameters:
Channel: HTTP session channel from where the configuration should be retrieved.
…: List of options supported for this API.
Following are the options supported by HTTP session channel:

- WIP_COPT_END: To mark the termination of the parameter list.
- **WIP_COPT_SND_BUFSIZE:** The current size of the TCP socket send buffer.
- W**IP_COPT_RCV_BUFSIZE:** The current size of the TCP socket receive buffer.
- WIP_COPT_PROXY_STRADDR: The hostname of the HTTP proxy server, a NULL value disables the proxy server.
- WIP_COPT_PROXY_PORT: The port number of the HTTP proxy server, the default value is 80.
- WIP_COPT_HTTP_VERSION: The HTTP version to be used by the session.
- Following are the options supported by HTTP data channel:
- WIP_COPT_END: To mark the termination of the parameter list.
- WIP_COPT_HTTP_STATUS_CODE: The 3-digit status code of the response.
- WIP_COPT_HTTP_STATUS_REASON: The reason phrase of the response, the first value points to the buffer where the reason phrase is to be written, the second value is the size of the buffer.
- WIP_COPT_HTTP_HEADER: The value of the HTTP message header field with the name given by the first value, the second value points to the buffer where the field value is to be written, the third value is the size of the buffer.

Returned Values:
0 is returned on success.
In case of an error, the various negative error codes that are returned are described below:
WIP_CERR_NOT_SUPPORTED: This channel does not support this operation.
WIP_CERR_INVALID: Invalid option.
WIP_CERR_CSTATE: The channel is not ready for retrieving the configuration. This will be returned either when channel is still initializing or it is already closed.
Aborting a HTTP session Channel
Internet Plug-In provide an API wip_abort() to abort and close the HTTP session channel. Note that this API is supported only by HTTP session channel.

Prototype:

```
s32 wip_abort (wip_channel_t Channel);
```

Parameters:
Channel: The HTTP session channel which has to be aborted.

Returned Values:
0 is returned on success.
In case of an error a negative value is returned.
WIP_CERR_NOT_SUPPORTED: This value is returned in case this API is used to abort either TCP server or UDP channel.
WIP_CERR_INTERNAL: This value is returned when there is a internal problems to abort the channel.
Closing HTTP Session/Data channel
Internet Plug-In provide an API wip_close() to close the HTTP session or data channel. When this API is used to close HTTP session channel, it releases all the resources associated with the session channel. Note that closing session channel will not close the data channel. The data channel should be closed explicitly by calling this function. Closing the data channel will make the session ready for another request.

Prototype:

```
s32 wip_close(wip_channel_t Channel)
```

Parameters:
Channel: HTTP session or data channel that should be closed.

Returned Values:
0 is returned on success.
In case of an error, it returns the following:
WIP_CERR_MEMORY: Insufficient memory
WIP_CERR_INVALID: Specified NULL channel

10. Sample Code

```
#include "adl_global.h"
#include "wip_net.h"
#include "wip_bearer.h"
#include "wip_http.h"

const u16 wm_apmCustomStackSize = 4096;

#define GPRS_APN      "airtelgprs.com"
#define GPRS_LOGIN    "airtel"
#define GPRS_PASSWORD "tango"

#define HTTP_PORT    80
const ascii * HTTP_STR_HOSTNAME   = "http://koivi.com/php-http-auth/protect.php";
const ascii * HTTP_STR_USERNAME   = "tester";
const ascii * HTTP_STR_PASSWORD   = "testing";
const ascii * HTTP_URLNAME = "http://koivi.com/php-http-auth/protect.php";

wip_bearer_t bearerHandle = NULL;
wip_channel_t httpCnxCh = NULL;
wip_channel_t httpDataCh = NULL;
adl_tmr_t *timer_ptr;
u32 timeout_period = 5;
```

```
void simHandler ( u8 Event );
void evh_bearer( wip_bearer_t b, s8 event, void *ctx);
void Timer_Handler (u8 Id, void *context);
void httpDataHandler(wip_event_t *ev, void *ctx );

void httpDataHandler(wip_event_t *ev, void *ctx )
{
switch( ev->kind)
{
case WIP_CEV_OPEN:
TRACE((1,"httpDataHandler: WIP_CEV_OPEN"));
break;
case WIP_CEV_READ:
{
u8 buf[256 ];
wm_memset(buf,0,256);

TRACE((1,"httpDataHandler: WIP_CEV_READ"));
while( wip_read(httpDataCh, buf, 256 ) > 0 )
{
TRACE((1, buf));
}
}
break;
case WIP_CEV_WRITE:
TRACE((1,"httpDataHandler: WIP_CEV_WRITE"));
break;
case WIP_CEV_PEER_CLOSE:
TRACE((1,"httpDataHandler: WIP_CEV_PEER_CLOSE"));
wip_close(httpDataCh);
wip_close(httpCnxCh);
break;
case WIP_CEV_ERROR:
TRACE((1,"httpDataHandler: WIP_CEV_ERROR"));
break;
}
}

void Timer_Handler (u8 Id, void *context)
{
TRACE ((1,"Inside timer handler. Timer id is %d", Id));
/* Start the HTTP Data session */
httpDataCh =  wip_getFileOpts(httpCnxCh,
HTTP_URLNAME,
httpDataHandler,
NULL,
WIP_COPT_HTTP_METHOD,
WIP_HTTP_METHOD_GET,
WIP_COPT_END);
TRACE (( 1, "wip_getFileOpts: %x",httpDataCh ));
}

void evh_bearer( wip_bearer_t b, s8 event, void *ctx)
{
switch(event)
{
case WIP_BEV_IP_CONNECTED:
TRACE (( 1, "evh_bearer: <WIP_BEV_IP_CONNECTED>"));
/* Start the HTTP session */
httpCnxCh = wip_HTTPClientCreateOpts( NULL, NULL,
WIP_COPT_HTTP_PROXY_PORT,HTTP_PORT,
WIP_COPT_USER,HTTP_STR_USERNAME,
WIP_COPT_PASSWORD,HTTP_STR_PASSWORD,
WIP_COPT_END);
TRACE (( 1, "wip_HTTPClientCreateOpts: %x",httpCnxCh ));

timer_ptr = (adl_tmr_t*) adl_tmrSubscribe (FALSE, timeout_period,
ADL_TMR_TYPE_100MS, (adl_tmrHandler_t) Timer_Handler);
break;
case WIP_BEV_CONN_FAILED:
TRACE (( 1, "evh_bearer: <WIP_BEV_CONN_FAILED>"));
wip_bearerClose( bearerHandle );
break;
case WIP_BEV_STOPPED:
TRACE (( 1, "evh_bearer: <WIP_BEV_STOPPED>"));
wip_bearerClose(bearerHandle );
break;
default:
TRACE (( 1, "evh_bearer: <Other Events>"));
break;
}
}

void simHandler ( u8 Event )
{
if( ADL_SIM_EVENT_FULL_INIT == Event )
{
s8 ret;
/* Open the GPRS bearer */
ret = wip_bearerOpen ( &bearerHandle, "GPRS", ( wip_bearerHandler_f ) evh_bearer, NULL);
TRACE (( 1, "wip_bearerOpen: ret = %d",ret ));
ret = wip_bearerSetOpts( bearerHandle,
WIP_BOPT_GPRS_APN, GPRS_APN,
WIP_BOPT_LOGIN,    GPRS_LOGIN,
WIP_BOPT_PASSWORD, GPRS_PASSWORD,
WIP_BOPT_END);
```

```
TRACE (( 1, "wip_bearerSetOpts: ret = %d",ret ));
ret = wip_bearerStart( bearerHandle );
}
}

void adl_main (adl_InitType_e adlInitType)
{
/* Subscribe to the SIM events */
adl_simSubscribe (( adl_simHdlr_f ) simHandler, NULL);

/* Initialize the stack */
wip_netInit();
}
```

Summary

The following points have been covered in this section

- **HTTP APIs are used for creating HTTP session and data channel.**
- **Following APIs are used for creating HTTP session channel**
  - **wip_HTTPClientCreate (creating a HTTP session channel with default parameter)**
  - **wip_HTTPClientCreateOpts (creating a HTTP session channel with userd defined parameters)**
- **Following APIs are used for creating HTTP data channel**
  - **wip_getFile (creating a data channel with default pa rameters for downloading file)**
  - **wip_getFileOpts (creating a data channel with user defined parameters for downloading file)**
  - **wip_putFile (creating a data channel with default parameters for uploading file)**
  - **wip_putFileOpts (creating a data channel with user defined parameters for uploading file)**
- **wip_getOpts API is used for retrieving the configuration parameters from the HTTP session channel**
- **wip_setOpts API is used for configuring the parameters of the HTTP session channel**
- **wip_read API is used for reading file data from the HTTP data channel**
- **wip_write API is used for writing file data to the HTTP data channel**
- **wip_shutdown API is used for shutting down the HTTP data channel**
- **wip_abort API is used for aborting the HTTP data channel**
- **wip_close API is used for closing both HTTP session and data channel**

APPENDIX

Abbreviations and Acronyms

The following table lists the abbreviations that you will encounter in this document:

Abbreviation/Acronym	Description
A&D	Application And Data Storage
ADL	Application Development Layer
ADC	Analog to Digital Converter
APDU	Application Protocol Data Unit
API	Application Programming Interface
APN	Access Point Name
ARPANET	Advanced Research Projects Agency NETwork (U.S. Department of Defense)
ASCII	American Standard Code for Information Interchange
AT	Attention
CGPS	Companion Global Positioning System
Cid	Context Identifier
CLIP	Caller Line Identification Presentation
CHAP	Challenge-Handshake Authentication Protocol
CTS	Clear To Send signal
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DOTA	Download Over The Air
DSR	Data Set Ready
DTMF	Dual Tone Multi Frequency
EDGE	Enhanced Data for Global Evolution
EEPROM	Electrically Erasable Programmable Read Only Memory
EXTINT	External Interruption
FCM	Flow Control Manager
FIFO	First In First Out
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
GPI	General Purpose Input
GPIO	General Purpose Input Output
GPO	General Purpose Output
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile communication
HTTP	Hyper Text Transfer Protocol
I2C	Inter Integrated Circuit
IDE	Integrated Development Environment
IP	Internet Protocol

Abbreviation/Acronym	Description
IO	Input Output
JRE	Java Runtime Environment
LIFO	Last In First Out
LSB	Least Significant Byte
MO	Mobile Originated
MS	Mobile Station
MSB	Most Significant Byte
MSCHAP v1	Microsoft Challenge Handshake Authentication Protocol version 1
MSCHAP v2	Microsoft Challenge Handshake Authentication Protocol version 2
MT	Mobile Terminated
OASiS	Open AT Embedded Software Suite
OS	Operating System
OSA	Open Sim Access
OSI	Open Systems Interconnect
PAP	Password Authentication Protocol
PASV	FTP command to enter passive mode
PC	Personal Computer
PDF	Portable Document Format
PDP	Packet data Protocol
PDU	Protocol Data Unit
PIN	Personal Identification Number
POP3	Post Office Protocol Version 3
PPP	Point–to–Point Protocol
PUK	Personal Unblocking Key
AirPrime Q24 Series	AirPrime Q24 Classic, AirPrime Q24Plus, AirPrime Q24 Extended and AirPrime Q24 Auto
RAM	Random Access Memory
RFC	Request for Comments
ROM	Read Only Memory
RTE	Remote Task Execution
RTOS	Real Time Operating System
SC	Service Centre
SDK	Software Development Kit
SGT	Software Generation Toolkit
SIM	Subscriber Identity Module
SME	Short Message Entity
SMS	Short Message Services
SMMT	Short Message Mobile Terminated
SMMO	Short Message Mobile Originated
SMTP	Simple Mail Transfer Protocol
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
TCU	Timer & Capture Unit
UART	Universal Asynchronous Receiver& Transmitter
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
URI	Universal Resource Identifier

Abbreviation/Acronym	Description
URL	Universal Resource Locator
USB	Universal Serial Bus
WAP	Wireless Application Protocol
WIP	Sierra Wireless Internet Plug-In

Glossary

The following table lists and describes the technical terms that you will find in this document.

Term	Description
Dual Tone Multi Frequency (DTMF)	The type of audio signals that are generated when the buttons on a touch-tone telephone are pressed. Pressing a button generates two simultaneous tones which are decoded by the exchange to determine which key is pressed.
Embedded application	Open AT Application sources to be compiled and run on a Sierra product.
Embedded software	Open AT Application binary: Set of Open AT Application and Open AT OS.
External application	Application external to Sierra product that sends AT commands through the serial port.
Gain	The amount of increase that an amplifier provides on the output side of the circuit.
GPS External mode	The GPS embedded module is driven by an external host. The GSM embedded module cannot control or detect GPS frames.
GPS Internal mode	The GPS embedded module is driven by the GSM embedded module.
Melody	A melody is a succession of tones of varying pitch forming a distinctive sequence.
Pre–parsing	Process for intercepting AT commands and AT responses.
Protocol Data Unit	The Protocol Data Unit contains the SMS content along with required header information as a part of it.
Remote Mode	An Open AT Application execution mode where the Open AT Application is executed on a simulated environment on the PC without actually downloading it on the target embedded module.
Short Message Entity	An entity which may send or receive Short Messages. The SME may be located in a fixed network, a mobile, or a SMSC (Short Messages Service Centre).
Short Message Service	The SMS (Short Messaging Service) that allows a SME (Short Message Entity) to send short text messages to other SME.
Short Message Service Centre	To allow SMS (Short Message Service) messages to be sent from your mobile phone you will require a SMSC number. This will instruct your mobile phone to send all SMS messages to this centre. The SMSC forwards the short message to the indicated destination subscriber number.
Target Mode	The Open AT Application is downloaded on the actual target (WISMO embedded module) and is executed in the real time environment.
Tone	Tone represents a sound of fixed pitch of a unique single frequency only.
Xmodem Protocol	File transfer protocol for transferring files over serial lines. XMODEM uses 128-byte packets with error detection, allowing the receiver to request retransmission of corrupted packets. XModem is fairly slow but reliable.
Embedded application	Open AT Application sources to be compiled and run on a Sierra Wireless product.
Firmware	A binary file which manages the lower level functionality (hardware and software (such as GSM and GPRS protocol)).
Internet Plug-In	Sierra Wireless Internet Plug-In is a Plug-In provided with Open AT OS which can be used for creating sockets, FTP, SMTP, HTTP and POP3 sessions.
Embedded module	Open AT OS compatible product supporting an Open AT Application.

Copyright

Trademarks

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	
Technical Support:	support@sierrawireless.com	
RMA Support:	repairs@sierrawireless.com	
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: http://www.sierrawireless.com

Special Thanks

Special thanks not only to the Open AT® training organizers Sylvain O., Catherine T., and Jacques S., but also to Infosys team for their constant help.