Muse PLATFORM™

DEVELOPMENT GUIDE

wavecom

MUSE PLATFORM™

DEVELOPMENT GUIDE

Version: 003
Date: April, 05th 2002
Reference: WM_SW_OAT_UGD_002

wavecom

Contents

4 FUNCTIONING 58

LIST OF FIGURES

1 Introduction

1.1 Purpose

This User's Guide describes the Open AT facility and provides guidelines for developing an Embedded Application.

1.2 References

I. Tools Manual
II. AT Command Interface Guide

1.3 Glossary

Application Mandatory API..Mandatory software interfaces to be used by the Embedded Application.

AT commands........................Set of standard modem commands.

AT function............................Software that processes the AT commands and AT subscriptions.

Embedded API layer..............Software developed by Wavecom, containing the Open AT APIs (Application Mandatory API, AT Command Embedded API, OS API, Standard API, FCM API, IO API, and BUS API).

Embedded Application.........User application sources to be compiled and run on a Wavecom product.

Embedded Core software....Software that includes the Embedded Application and the Wavecom library.

Embedded software..............User application binary: set of Embedded Application sources + Wavecom library.

External Application..............Application external to the Wavecom product that sends AT commands through the serial link.

Target....................................Open AT compatible product supporting an Embedded Application.

Target Monitoring Tool.........Set of utilities used to monitor a Wavecom product.

Receive command ***pre-parsing***.................Process for intercepting AT responses.

Send command ***pre-parsing***.....................Process for intercepting AT commands.

Standard API..........................Standard set of "C" functions.

Wavecom library...................Library delivered by Wavecom to interface Embedded Application sources with Wavecom Core Software functions.

Wavecom Core Software.....Set of GSM and open functions supplied to the User.

1.4 Abbreviations

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

2 DESCRIPTION

2.1 Software Architecture

2.1.1 Software Organization

The Open AT facility is a software mechanism. It relies on the following software architecture:

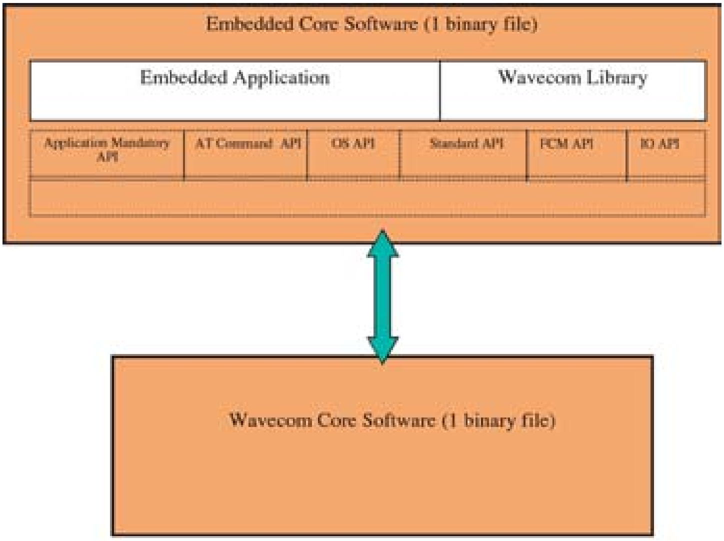

Embedded Core Software (1 binary file)
Embedded Application
Wavecom Library
Application Mandatory API
AT Command API
OS API
Standard API
FCM API
IO API
Wavecom Core Software (1 binary file)

Figure 1: General Software Architecture

The different software elements on a Wavecom product are described here-below.
The **Embedded Core Software** (binary file) includes the following items:

- ❑ the Embedded Application: application to be developed and downloaded into the Wavecom Target product. The Embedded Application must be linked to the Wavecom library.
- ❑ the Wavecom library: software library provided by Wavecom (included in the Open AT SDK) and based on the Embedded API layer.
- ❑ the Embedded API Layer (developed by Wavecom), which includes:
  - the Application Mandatory API: mandatory software interfaces to be used by the Embedded Application,
  - the AT Command API: software interfaces providing access to the set of AT functions,
  - the OS API: software interfaces providing access to the Operating System functions,
  - the FCM API: software interfaces providing access to the Flow Control Manager functions (secure access to V24 and Data IO flows),
  - the IO API: software interfaces providing control on the serial link mode, and on the Gpio devices.
  - the BUS API: software interfaces providing control on bus devices (as SPI or I2C bus).
  - the Standard API: standard set of “C” functions.
- ❑ The **Wavecom Core Software** (another binary file), manages the GSM protocol.

2.1.2 Software Supplied by Wavecom

The software items supplied are as follows:

- ❑ one software library, wmopenat.lib,
- ❑ one set of header files (.h), defining the Open AT API functions,
- ❑ source code samples,
- ❑ a set of tools called Development ToolKit, for designing and testing any application (see document [Ref I]).

2.2 Minimum Embedded Application Code

The following code must be included in any Embedded Application:

```
char wm_apmCustomStack[1024];
/* the value 1024 is an example */
const u16 wm_apmCustomStackSize = sizeof (wm_apmCustomStack];

s32 wm_apmAppliInit (wm_apmInitType_e  InitType)
{
   return OK;
}
s32 wm_apmAppliParser ( wm_apmMsg_t * Message )
{
   return OK;
}
```

wm_apmCustomStack and **wm_apmCustomStackSize** are two mandatory variables, used to define the application call stack size (see § 3.2.1: "Stack Initialization").
wm_apmAppliInit() is a mandatory function; this is the first function called at the embedded application initialization (see § 3.2.2: "The wm_apmAppliInit").
wm_apmAppliParser() is a mandatory function; it is called each time the embedded application receives a message from the Wavecom Core Software (see § 3.2.3: "The wm_apmAppliParser").

2.3 Specificity of AT Commands in the Open AT Architecture

See document [Ref II].

2.3.1 AT Command Size

The maximum size of an AT command string or a Response string that can be sent through the serial link is 512 bytes. Therefore, if the Embedded Application needs to send more data, it must be sent in several increments.

2.3.2 AT+WDWL Command

The AT+WDWL command, used to download an application, is not pre-parsed. Therefore, even if the Embedded Application has subscribed to the command pre-parsing mechanism, this command is processed by means of the Wavecom software and it is not sent back to this application.

Note: *the AT+WDWL command is described in the document [Ref II].*

2.3.3 AT+WOPEN Command

Open AT require some specific AT commands such as AT+WOPEN.
The latter is described below.
This command is always available for an External Application. It is not pre-parsed and it is treated even if the AT software is busy.
This command deactivates an Embedded Application in order to ensure that a new application can be downloaded. Typically, if an Embedded Application continuously sends AT commands, the Wavecom AT command software is always busy. Therefore, if the AT+WDWL command is sent by an External Application, it is not processed.
AT+WOPEN can take the values 0 (= Stop) and 1 (=Start):

- ❑ Sending the AT+WOPEN=0 command first, by means of an External Application, deactivates the Embedded Application: a new Embedded Application may then be downloaded.
- ❑ If the Embedded Application is deactivated, it can be restarted using AT+WOPEN=1. The module then reboots and this application is restarted 20 sec after the module boot.

Note: *Refer to the document [Ref II] for an overview of the complete set of AT commands.*

2.4 Notes on Memory Management

The Embedded software runs within an RTK task: the user must define the size of the customer application call stack.

The Wavecom Core Software and the Embedded application manage their own RAM area. Any access from one of these programs to the other's RAM area is prohibited and causes a reboot.

In case an Embedded Application uses more than the maximum allocated RAM in global variables, or uses more than the maximum allocated ROM, then the behavior of the Embedded software becomes erratic.

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

The application can use up to 32 KB of RAM, and 384 KB of ROM.

2.5 Known Limitations

2.5.1 Command Pre-Parsing Limitation

In normal operating mode, the target serial link manager checks to see whether every command starts with "AT" and ends with a carriage return + with a char string end. Therefore, the only commands to be dispatched to the Embedded Application (in case of command pre-parsing subscription) are the ones complying with the here-above description.

2.5.2 Missing Unsolicited Messages in Remote Application

In Remote Application Execution mode, the application is started a few seconds after the Target. Therefore, some unsolicited events might be lost.

A pre-processor flag like __REMOTETASKS__ can be used to add some specific code for remote mode.

2.6 Security

2.6.1 Software Security

Two software safeguards are used in the Open AT platform: RAM access protection and watchdog protection.
After reboot, the “**wm_apmAppliInit ()**” function will have the parameter set to WM_APM_REBOOT_FROM_EXCEPTION.
After reboot, the application is started only 20 seconds after the start of the Wavecom core software. This allows at least 20 seconds to re-download a new application.

2.6.1.1 RAM Access Protection

A specific RAM area is allocated to the Embedded Application.
The Embedded Application is seen as a Real-Time Task in the Wavecom software, and each time this task runs, the Wavecom RAM protection is activated.
If the Embedded Application tries to access this RAM, then an exception occurs and the software reboots.
In case of illegal RAM access, the Target Monitoring Tool screen displays:
”**ARM exception 1 xxx**”, where “xxx” is the address the application was attempting to access.
If the symbol file is correctly configured in the Target Monitoring Tool (see document [Ref I]), then a Back Trace must describe the affected C functions in which the crash occurred.

2.6.1.2 Watchdog Protection

The Wavecom Core software is protected from reaching a dead-end lock by a 5-second watchdog.
To ensure that the embedded application is not the cause of the crash, there is a specific 4.5-second watchdog of the embedded application, so an embedded application crash can be detected.
In case of a crash, the software reboots.
If an embedded application crash is detected, the Target Monitoring Tool screen displays:
”**Customer watchdog**”.

2.6.2 Hardware Security

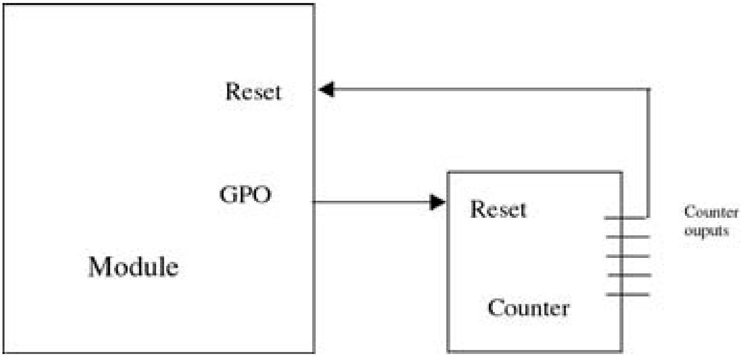
Protection can also be improved using an external watchdog reset circuitry.

With such a hardware watchdog protection, the Wavecom product will always be reset even in case of the software crashes.

To achieve this, one can use a GPO connected to a specific hardware counter that will reset the product if not refreshed.

For example, this specific hardware can be a counter with a specific counter output connected to the reset pin of the module, and the counter reset pin connected to a GPO.

In this way, the software in the module is supposed to reset the counter periodically. If not, the counter will increase until it reaches the specified limit and then resets the module.



3 API

3.1 Data Types

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

3.2 Mandatory Functions

The API described below includes a set of functions the Embedded software must supply and some mandatory variables the Embedded software must set.
This API is located in the wm_apm.h file.

3.2.1 Stack Initialization

The following mandatory variables are used to define the stack size:

```
char wm_apmCustomStack[1024];              /* the value 1024 is an example */
const u16 wm_apmCustomStackSize = sizeof(wm_apmCustomStack);
```

These data represent the amount of memory needed by the customer call stack.

3.2.2 The wm_apmAppliInit Function

wm_apmAppliInit function is called just once during initialization.
Its prototype is:

```
s32 wm_apmAppliInit ( wm_apmInitType_e  InitType );
```

3.2.2.1 Parameter

InitType:
Works out the item that triggered the initialization. The corresponding values are:

```
typedef enum
{
        WM_APM_POWER_ON,
        WM_APM_REBOOT_FROM_EXCEPTION
}       wm_apmInitType_e;
```

WM_APM_POWER_ON means that normal Power On has occurred.
WM_APM_REBOOT_FROM_EXCEPTION means the module has restarted after an exception.
The following events may cause an exception:

- ❑ a call to the **wm_osDebugFatalError()** function,
- ❑ unauthorized RAM access,
- ❑ a customer task watchdog.

3.2.2.2 Required Header

```
Wm_apm.h
```

3.2.2.3 Return Value

The returned value is not relevant

3.2.3 The wm_apmAppliParser Function

This function is called whenever a message is received from the Wavecom Core Software. Its prototype is:

```
s32 wm_apmAppliParser ( wm_apmMsg_t * Message );
```

3.2.3.1 Parameter

Message:

The *Message* structure depends on its type:

```
typedef struct
{
        s16             MsgTyp;         /* Type of the received message:
                                        works out the associated structure of
                                        the message body part*/
        wm_apmBody_t    Body;           /* Specific message body */
} wm_apmMsg_t;
```

MsgTyp may have the following values:

- ❑ **WM_AT_RESPONSE** means the message includes an AT command response sent by the Embedded Application.
- ❑ **WM_AT_UNSOLICITED** means the message includes an unsolicited AT response.
- ❑ **WM_AT_INTERMEDIATE** means the message includes an intermediate AT response.
- ❑ **WM_AT_CMD_PRE_PARSER** means the message includes an AT command sent by the External Application.
- ❑ **WM_AT_RSP_PRE_PARSER** means the message includes a response processed by a Wavecom Core Software AT function.
- ❑ **WM_OS_TIMER** means the message is sent when the timer expires.
- ❑ **WM_OS_RELEASE_MEMORY** means the message includes the address of a released pointer.
- ❑ **WM_FCM_RECEIVE_BLOCK** means the message includes data received by the embedded application.
- ❑ **WM_FCM_OPEN_FLOW** means the requested flow opening operation is successful.
- ❑ **WM_FCM_CLOSE_FLOW** means the requested flow closing operation is successful.
- ❑ **WM_FCM_RESUME_DATA_FLOW** means the embedded application may resume its data sending operations.
- ❑ **WM_IO_SERIAL_SWITCH_STATE_RSP** includes the response to the serial link mode switching request.

The body structure is given below:

```
typedef union
{
        /* Includes herein the different specific structures associated to   MsgTyp */
        /* WM_AT_RESPONSE                                                      */
        wm_atResponse_t                                                        ATResponse;
        /* WM_AT_UNSOLICITED                                                   */
        wm_atUnsolicited_t                                                     ATUnsolicited;
        /* WM_AT_INTERMEDIATE                                                  */
        wm_atIntermediate_t                                                    ATIntermediate;
        /* WM_AT_CMD_PRE_PARSER                                                */
        wm_atCmdPreParser_t                                                    ATCmdPreParser;
        /* WM_AT_RSP_PRE_PARSER                                                */
        wm_atRspPreParser_t                                                    ATRspPreParser
        /* WM_OS_TIMER                                                         */
        wm_osTimer_t                                                           OSTimer;
        /* WM_OS_RELEASE_MEMORY                                                */
        wm_osRelease_t                                                         OSRelease;
        /* WM_FCM_RECEIVE_BLOCK                                                */
        wm_fcmReceiveBlock_t                                                   FCMReceiveBlock;
        /* WM_FCM_OPEN_FLOW                                                    */
        wm_fcmOpenFlow_t                                                       FCMOpenFLow
        /* WM_FCM_CLOSE_FLOW                                                   */
        wm_fcmFlow_e                                                           FCMCloseFlow
        /* WM_FCM_RESUME_DATA_FLOW                                             */
        wm_fcmFlow_e                                                           FCMResumeFlow
        /* WM_IO_SERIAL_SWITCH_STATE_RSP                                       */
        wm_ioSerialSwitchStateRsp_t
        IOSerialSwitchStateRsp
} wm_apmBody_t;
```

The sub-structures of the message body are listed below:

Body for WM_AT_RESPONSE:

```
typedef struct          {
        wm_atSendRspType_e    Type;
        u16                               StrLength; /* Length of StrData[] */
        ascii                                       StrData[1];/* AT response */
} wm_atResponse_t;
typedef enum            {
        WM_AT_SEND_RSP_TO_EMBEDDED,
        WM_AT_SEND_RSP_TO_EXTERNAL,
        WM_AT_SEND_RSP_BROADCAST
} wm_atSendRspType_e;
```

(See § 3.3.1: “The wm_atSendCommand” for wm_atSendRspType_e description).

Body for WM_AT_UNSOLICITED:

```
typedef struct          {
        wm_atUnsolicited_e      Type;
        u16                               StrLength;
        ascii                                       StrData[1];
} wm_atUnsolicited_t;
typedef enum            {
        WM_AT_UNSOLICITED_TO_EXTERNAL,
        WM_AT_UNSOLICITED_TO_EMBEDDED,
        WM_AT_UNSOLICITED_BROADCAST
} wm_atUnsolicited_e;
```

(See § 3.3.2: “The wm_atUnsolicitedSubscription ” for wm_atUnsolicited_e description).

Body for WM_AT_INTERMEDIATE:

```
typedef struct          {
        wm_atIntermediate_e   Type;
        u16             StrLength;
        ascii                   StrData[1];
} wm_atIntermediate_t;
typedef enum            {
        WM_AT_INTERMEDIATE_TO_EXTERNAL,
        WM_AT_INTERMEDIATE_TO_EMBEDDED,
        WM_AT_INTERMEDIATE_BROADCAST
} wm_atIntermediate_e;
```

(See § 3.3.3: “The wm_atIntermediateSubscription” for wm_atIntermediate_e description).

Body for WM_AT_CMD_PRE_PARSER:

```
typedef struct          {
        wm_atCmdPreSubscribe_e          Type;
        u16                             StrLength;
        ascii                                   StrData[1];
} wm_atCmdPreParser_t;
typedef enum                    {
        WM_AT_CMD_PRE_WAVECOM_TREATMENT, /* Default value */
        WM_AT_CMD_PRE_EMBEDDED_TREATMENT,
        WM_AT_CMD_PRE_BROADCAST
} wm_atCmdPreSubscribe_e;
```

(See § 3.3.4: "The wm_atCmdPreParserSubscribe" for wm_atCmdPreSubscribe_e description).

Body for WM_AT_RSP_PRE_PARSER:

```
typedef struct          {
        wm_atRspPreSubscribe_e                  Type;
        u16                             StrLength;
        ascii                                   StrData[1];
} wm_atRspPreParser_t;
typedef enum                    {
        WM_AT_RSP_PRE_WAVECOM_TREATMENT,  /* Default value */
        WM_AT_RSP_PRE_EMBEDDED_TREATMENT,
        WM_AT_RSP_PRE_BROADCAST
} wm_atRspPreSubscribe_e;
```

(See § 3.3.5: "wm_atRspPreParserSubscribe" for wm_atRspPreSubscribe_e description).

Body for WM_OS_TIMER:

```
typedef struct          {
        u8  Ident;      /* Timer identifier */
} wm_osTimer_t;
```

(See § 3.4.1: "The wm_osStartTimer" for timer identifier description).

Body for WM_OS_RELEASE_MEMORY:

```
typedef struct          {
        void                            *pMemoryBlock;
} wm_osRelease_t;
```

(See § 3.5.3: "The wm_fcmSubmitData" for this message description).

Body for WM_FCM_RECEIVE_BLOCK:

```
typedef struct          {
        u16             DataLength;         /* number of bytes received */
        u8              Reserved1[2];
        wm_fcmFlow_e            FlowId;             /* IO flow ID */
        u8              Reserved2[7];
        u8              Data[1];            /* data received */
} wm_fcmReceiveBlock_t;

typedef enum                    {
        WM_FCM_DATA,
        WM_FCM_V24
} wm_fcmFlow_e;
```

(See § 3.5.4: “Receive Data Blocks” for wm_fcmReceiveBlock_t description).

Body for WM_FCM_OPEN_FLOW:

```
typedef struct          {
        wm_fcmFlow_e            FlowId;                         /* opened IO flow ID */
        u16             DataMaxToSend;              /* max length of sent data */
} wm_fcmOpenFlow_t;

typedef enum                    {
        WM_FCM_DATA,
        WM_FCM_V24
} wm_fcmFlow_e;
```

(See § 3.5.1: “The wm_fcmOpenDataAndV24” for wm_fcmOpenFlow_t description).

Body for WM_FCM_CLOSE_FLOW:

```
typedef enum                    {
        WM_FCM_DATA,
        WM_FCM_V24
} wm_fcmFlow_e;
```

(See § 3.5.2: “The wm_fcmCloseDataAndV24” for wm_fcmFlow_e description).

Body for WM_FCM_RESUME_DATA_FLOW:

```
typedef enum                    {
        WM_FCM_DATA,
        WM_FCM_V24
} wm_fcmFlow_e;
```

(See § 3.5.3: “The wm_fcmSubmitData” for wm_fcmFlow_e description).

Body for WM_IO_SERIAL_SWITCH_STATE_RSP:

```
typedef struct                  {
        wm_ioSerialSwitchState_e        SerialMode;         /* mode requested */
        s8                              RequestReturn;      /* <0 means error */
} wm_ioSerialSwitchStateRsp_t;
```

(See § 3.6.1.1: “The wm_ioSerialSwitchState Fonction” for wm_ioSerialSwitchStateRsp_t description).

3.2.3.2 Return Values

The return parameter indicates whether the message has been taken into account (OK : 0) or not (ERROR : -1).

3.2.3.3 Required Header

Wm_apm.h

3.2.3.4 Notes

- ❑ any **StrData[]** or **Data[]** parameter present in the body sub-structure is automatically released at the end of the function.
- ❑ any **StrData[]** data is terminated by a 0x00 character and any associated **StrLength** includes the 0x00 character.

3.3 AT Command API

3.3.1 The wm_atSendCommand Function

The wm_atSendCommand function sends AT commands.
Its prototype is:

```
void  wm_atSendCommand
(                          u16                          AtStringSize,
                           wm_atSendRspType_e           ResponseType,
                           ascii                        *AtString );
```

3.3.1.1 Parameters

AtString:

Any AT command string in ASCII character (terminated by a 0x00). Many strings can be sent at the same time, depending on the type of AT command.

AtStringSize:

Size of the previous parameter, **AtString**. It equals the length + 1 and includes the 0x00 character.

ResponseType:

Indicates which application receives the AT responses. The corresponding values are:

```
typedef enum                {
      WM_AT_SEND_RSP_TO_EMBEDDED,                /* Default value */
      WM_AT_SEND_RSP_TO_EXTERNAL,
      WM_AT_SEND_RSP_BROADCAST
} wm_atSendRspType_e;
```

WM_AT_SEND_RSP_TO_EMBEDDED means that all the AT responses will be sent back to the Embedded Application (default mode).
WM_AT_SEND_RSP_TO_EXTERNAL means that all the AT responses will be sent back to the External Application (PC).
WM_AT_SEND_RSP_BROADCAST means that all the AT responses will be broadcasted to both the Embedded and External Applications (PC).

3.3.1.2 Required Header

```
Wm_at.h
```

3.3.1.3 Notes

- ❑ As described in the “AT Commands Interface” document, AT commands sent by **wm_atSendCommand()** begin with the “AT” string, and end with a “\r” character (carriage return), except in some cases (“A/” command, SMS writing commands (“test\x1A”), …)
- ❑ AT Command responses are received by the Embedded Application through a message. This message is available as a parameter of the **wm_apmAppliParser()** function with the *MsgTyp* parameter set to WM_AT_RESPONSE (see § 3.2.3: “The wm_apmAppliParser”).
- ❑ A response sent to an External Application cannot be pre-parsed (see § 3.3.5: “wm_atRspPreParserSubscribe”). If an Embedded Application wants to filter or spy the response, it must set the *ResponseType* parameter to WM_AT_SEND_RSP_TO_EMBEDDED or WM_AT_SEND_RSP_BROADCAST.

3.3.1.4 Example: Sending AT Commands and Receiving the Corresponding Responses

The Embedded Application sends an AT command and receives the response from the AT functionality of Wavecom Core Software using The wm_atSendCommand and The wm_atSendRspExternalApp functions.

❑ An example of sending an AT command is given below:

```
wm_atSendCommand( 16, WM_AT_SEND_RSP_TO_EMBEDDED, “ATD0146290800\r” );
```

❑ An example of receiving an AT response is given below:

```
s32 wm_apmAppliParser (wm_apmMsg_t * Message)
{
    ascii *     strBuffer;
    u16         nLenBuffer;
    switch (Message->MsgTyp)
    {
        ....
        case WM_AT_SEND_RSP:
            strBuffer    = &(Message->Body.AT_Response.StrData);
            nLenBuffer = Message->Body. AT_Response.StrLength;
            /* Receive AT response for filtering        */
        if (Message->Body.ATResponse.Type == AT_RESPONSE_TO_EMBEDDED)
        {
                if (wm_strnicmp(strBuffer, “CONNECT”, 7) == 0)
            {
                    /* Local processing */
                    ....
                    wm_atSendRspExternalApp(“CONNECT\r”, 9);
                }
            else
            {
                    /* Don’t modify other responses */
                    wm_atSendRspExternalApp ( wm_strlen(strBuffer),
                                            strBuffer);
                }
            }
            /*  Receive AT response for spying          */
            else if (Message->Body.ATResponse.Type ==
                        WM_AT_SEND_RSP_BROADCAST)
                    { ...
             }
                /*  ERROR  */
             else
                { ..
             }
        ...
    }
    return OK;
}
```

3.3.2 The wm_atUnsolicitedSubscription Function

If the Embedded Application wants to receive an unsolicited AT response (incoming call, etc.), the wm_atUnsolicitedSubscription function is used to subscribe to the corresponding service.

Its prototype is:

```
void     wm_atUnsolicitedSubscription (
                                   wm_atUnsolicited_e  Unsolicited);
```

3.3.2.1 Parameter

Unsolicited:

Indicates which application receives the unsolicited AT response. The corresponding values are:

```
typedef enum                    {
        WM_AT_UNSOLICITED_TO_EXTERNAL,        /* Default value */
        WM_AT_UNSOLICITED_TO_EMBEDDED,
        WM_AT_UNSOLICITED_BROADCAST,
} wm_atUnsolicited_e;
```

WM_AT_UNSOLICITED_TO_EXTERNAL means any unsolicited AT response will be sent back to the External Application (PC). This is the default mode.

WM_AT_UNSOLICITED_TO_EMBEDDED means any unsolicited AT response will be sent back to the Embedded Application.

WM_AT_UNSOLICITED_BROADCAST means any unsolicited AT response will be broadcast to both the Embedded and External Applications (PC).

3.3.2.2 Required Header

```
Wm_at.h
```

3.3.2.3 Note

An unsolicited AT response is received by the Embedded Application through a message. This message is available as a parameter of the wm_apmAppliParser() function with *MsgTyp* parameter set to WM_AT_UNSOLICITED
(see § 3.2.3: “The wm_apmAppliParser”).

3.3.2.4 Example: Receiving Unsolicited AT Responses

The following example deals with The wm_atUnsolicitedSubscription function.
The two stages used to receive unsolicited AT responses are:

❶ Subscribing to an Embedded Application to receive unsolicited AT responses. Three types of subscriptions are available: default (WM_AT_UNSOLICITED_TO_EXTERNAL), filtering (WM_AT_UNSOLICITED_TO_EMBEDDED) and spying (WM_AT_UNSOLICITED_BROADCAST).

An example of a filter subscription is given below:

```
/* Unsolicited responses are process by Embedded Application */
wm_atUnsolicitedSubscription (WM_AT_UNSOLICITED_TO_EMBEDDED);
```

❷ Receiving unsolicited AT responses:

```
s32 wm_apmAppliParser (wm_apmMsg_t * Message)
{
        ascii *     strBuffer;
        u16         nLenBuffer;

        switch (Message->MsgTyp)
        {
                ....
                case WM_AT_UNSOLICITED:
                        strBuffer    = &(Message->Body.ATUnsolicited.StrData);
                        nLenBuffer = Message->Body.ATUnsolicited.StrLength;

                        /*  Process unsolicited AT response for filtering       */
                        if (Message->Body.ATUnsolicited.Type ==
                                        WM_AT_UNSOLICITED_TO_EMBEDDED)
                        {
                                /* Embedded processings */
                        }

                        /*  Process unsolicited AT response for spying          */
                        else if (Message->Body.ATUnsolicited.Type ==
                                        WM_AT_UNSOLICITED_BROADCAST)
                        {
                                /* Embedded processings */
                        }
                ...
        }
        return OK;
}
```

3.3.3 The wm_atIntermediateSubscription Function

If the Embedded Application wants to receive an intermediate AT response (alerting the remote party during a mobile-originated call, SMS reading responses, etc.), the wm_atIntermediateSubscription function is used to subscribe to the corresponding service.

Its prototype is:

```
void    wm_atIntermediateSubscription (
                                    wm_atIntermediate_e  Intermediate );
```

3.3.3.1 Parameter

Intermediate:

Indicates which application receives the intermediate AT response.

The corresponding values are:

```
typedef enum                    {
        WM_AT_INTERMEDIATE_TO_EXTERNAL,     /* Default value */
        WM_AT_INTERMEDIATE_TO_EMBEDDED,
        WM_AT_INTERMEDIATE_BROADCAST,
} wm_atIntermediate_e;
```

WM_AT_INTERMEDIATE_TO_EXTERNAL means any intermediate AT response will be sent back to the External Application (PC). This is the default mode.

WM_AT_INTERMEDIATE_TO_EMBEDDED means any intermediate AT response will be sent back to the Embedded Application.

WM_AT_INTERMEDIATE_BROADCAST means any intermediate AT response will be broadcasted to both the Embedded and External Applications (PC).

3.3.3.2 Required Header

```
Wm_at.h
```

3.3.3.3 Note

An intermediate AT response is received by the Embedded Application through a message. This message is available as a parameter of the wm_apmAppliParser() function with *MsgTyp* parameter set to WM_AT_INTERMEDIATE
(see § 3.2.3: “The wm_apmAppliParser”).

3.3.3.4 Example: Receiving Intermediate AT Responses

The following example deals with the wm_atIntermediateSubscription function.
The two stages which are used to receive intermediate AT responses are:
❸ Subscribing to an Embedded Application to receive intermediate AT responses. Three types of subscriptions are available: default (WM_AT_INTERMEDIATE_TO_EXTERNAL), filtering (WM_AT_INTERMEDIATE_TO_EMBEDDED) and spying (WM_AT_INTERMEDIATE_BROADCAST).
An example of a filter subscription is given below:

```
/* Intermediate responses are processed by Embedded Application */
wm_atIntermediateSubscription (WM_AT_INTERMEDIATE_TO_EMBEDDED);
```

❹ Receiving intermediate AT responses:

```
s32 wm_apmAppliParser (wm_apmMsg_t * Message)
{
        ascii *     strBuffer;
        u16         nLenBuffer;

        switch (Message->MsgTyp)
        {
                ....
                case WM_AT_INTERMEDIATE:
                        strBuffer    = &(Message->Body.ATIntermediate.StrData);
                        nLenBuffer = Message->Body.ATIntermediate.StrLength;

                        /*  Process intermediate AT response for filtering     */
                        if (Message->Body.ATIntermediate.Type ==
                                        WM_AT_INTERMEDIATE_TO_EMBEDDED)
                        {
                                /* Embedded processing */
                        }

                        /*  Process intermediate AT response for spying        */
                        else if (Message->Body.ATIntermediate.Type ==
                                        WM_AT_INTERMEDIATE_BROADCAST)
                        {
                                /* Embedded processing */
                        }
                ...
        }
        return OK;
}
```

3.3.4 The wm_atCmdPreParserSubscribe Function

If the Embedded Application wants to perform AT command pre-parsing, it should then subscribe to the corresponding services, using the wm_atCmdPreParserSubscribe function.

The AT messages received from the External Application are forwarded to the Pre-parser and sent to the Embedded Application through a WM_AT_CMD_PRE_PARSER type message, of which the associated structure is wm_atCmdPreParser_t.

Note that the "AT+WDWL" and "AT+WOPEN" AT commands are not pre-parsed, so that the User can download a new Embedded software whenever s/he wants.

The prototype of this function is:

```
void    wm_atCmdPreParserSubscribe (
                                wm_atCmdPreSubscribe_e   SubscribeType);
```

3.3.4.1 Parameter

SubscribeType:

Indicates what happens when an AT command arrives. The corresponding values are:

```
typedef enum                    {
        WM_AT_CMD_PRE_WAVECOM_TREATMENT, /* Default value */
        WM_AT_CMD_PRE_EMBEDDED_TREATMENT,
        WM_AT_CMD_PRE_BROADCAST
} wm_atCmdPreSubscribe_e;
```

WM_AT_CMD_PRE_WAVECOM_TREATMENT means the Embedded Application does not want to filter or spy the commands sent by an External Application (default mode).

WM_AT_CMD_PRE_EMBEDDED_TREATMENT means the Embedded Application wants to filter the AT commands sent by an External Application.

WM_AT_CMD_PRE_BROADCAST means the Embedded Application wants to spy the AT commands sent by an External Application.

3.3.4.2 Required Header

```
Wm_at.h
```

3.3.4.3 Notes

- ❑ Filtered or spied AT commands are received by the Embedded Application through a message. This message is available as a parameter of the **wm_apmAppliParser()** function with the *MsgTyp* parameter set to WM_AT_CMD_PRE_PARSER (see § 3.2.3: "The wm_apmAppliParser").
- ❑ The Embedded Application will process the received command and, for instance, will send it back either completely or not to the **wm_atSendCommand()** function. Therefore, the responses may be forwarded to the Wavecom Core Software.
- ❑ When a command is pre-parsed for filtering, the User has the responsibility to send the response to the External Application.

3.3.4.4 Example: Filtering or Spying AT Commands Sent by an External Application

The following example deals with the **wm_atCmdPreParserSubscribe()** function.

The two stages which are used to filter or spy AT commands sent by an External Application are:

❶ Subscribing to a command pre-parsing mechanism to filter or spy the AT commands sent by the External Application.

An example of a filtering subscription is given below:

```
/* Filter subscription */
wm_atCmdPreParserSubscribe(WM_AT_CMD_PRE_EMBEDDED_TREATMENT);
```

An example of a spying subscription is given below:

```
/* Spy subscription */
wm_atCmdPreParserSubscribe(WM_AT_CMD_PRE_BROADCAST);
```

❷ Receiving and processing the pre-parsed commands (an AT command sent by the External Application) in the Embedded Application:

```
s32 wm_apmAppliParser (wm_apmMsg_t * Message)
{
        ascii *     strBuffer;
        u16         nLenBuffer;

        switch (Message->MsgTyp)
        {
                ....
                case WM_AT_CMD_PRE_PARSER:
                        strBuffer    = &(Message->Body.ATCmdPreParser.StrData);
                        nLenBuffer = Message->Body. ATCmdPreParser.StrLength;

                        /*  Process pre-parsed AT command for filtering       */
                        if      (Message->Body.ATCmdPreParser.Type ==
                                        WM_AT_CMD_PRE_EMBEDDED_TREATMENT)
                        {
                                /* Filtering Embedded processings */
                                ...
                        }
                        else if (Message->Body.ATCmdPreParser.Type ==
                                         WM_AT_CMD_PRE_BRAODCAST)
                                 {
                                  /* Spying Embedded processing */
                                  ...
                          }
                ...
        }
        return OK;
}
```

3.3.5 The wm_atRspPreParserSubscribe Function

If the Embedded Application wants to perform an AT response pre-parsing, it should then subscribe to the corresponding services, using the wm_atRspPreParserSubscribe function. An AT message sent by an external application and processed by the Wavecom Core Software generates a response. Depending on the subscription type, this response may be forwarded to the Embedded Application through a message of the WM_AT_RSP_PRE_PARSER type of which the associated structure is wm_atRspPreParser_t. Its prototype is:

```
void    wm_atRspPreParserSubscribe (
                                  wm_atRspPreSubscribe_e   SubscribeType );
```

3.3.5.1 Parameter

SubscribeType:

Indicates what happens when an AT response arrives. The corresponding values are as follows:

```
typedef enum                    {
        WM_AT_RSP_PRE_WAVECOM_TREATMENT, /* Default value */
        WM_AT_RSP_PRE_EMBEDDED_TREATMENT,
        WM_AT_RSP_PRE_BROADCAST
} wm_atRspPreSubscribe_e;
```

WM_AT_RSP_PRE_WAVECOM_TREATMENT means the Embedded Application does not want to filter or spy the responses sent to an External Application (default mode).
WM_AT_RSP_PRE_EMBEDDED_TREATMENT means the Embedded Application wants to filter the AT responses sent to an External Application.
WM_AT_RSP_PRE_BROADCAST means the Embedded Application wants to spy the AT responses sent to an External Application.

3.3.5.2 Required Header

```
Wm_at.h
```

3.3.5.3 Notes

- ❑ Filtered or spied AT responses are received by the Embedded Application through a message. This message is available as a parameter of the **wm_apmAppliParser()** function with the *MsgTyp* parameter set to WM_AT_RSP_PRE_PARSER (see § 3.2.3: “The wm_apmAppliParser”).
- ❑ If the Embedded Application subscribes to WM_AT_RSP_PRE_EMBEDDED_TREATMENT, it will process the response and send it to the External Application, using the **wm_atSendRspExternalApp()** function (see § 3.3.6: “The wm_atSendRspExternalApp”).
- ❑ The response pre-parser will only be active if the AT command has not been sent through **wm_atSendCommand()**. In this case, the response is processed as described in the *ResponseType* parameter (see § 3.3.1: “wm_atSendCommand”).

3.3.5.4 Example: Filtering or Spying AT Responses Sent to the External Application

The following example deals with the wm_atRspPreParserSubscribe() function.

The two stages used to filter or spy the AT response sent to the External Application are:

❶ Subscribing to the response pre-parsing mechanism in order to filter or spy the AT response sent to the External Application.

An example of a filter subscription is given below:

```
/* Filter subscription */
wm_atRspPreParserSubscribe(WM_AT_RSP_PRE_EMBEDDED_TREATMENT);
```

An example of a spying subscription is given below:

```
/* Spy subscription */
wm_atRspPreParserSubscribe(WM_AT_RSP_PRE_BROADCAST);
```

❷ Processing the pre-parsed response in the Embedded Application:

```
s32 wm_apmAppliParser (wm_apmMsg_t * Message)
{
        ascii *     strBuffer;
        u16         nLenBuffer;

        switch (Message->MsgTyp)
        {
                ....
                case WM_AT_RSP_PRE_PARSER:
                        strBuffer = &(Message->Body.ATRspPreParser.StrData);
                        nLenBuffer = Message->Body.ATRspPreParser.StrLength;

                        /*  Process pre-parsed AT command for filtering      */
                                if(Message>Body.ATRspPreParser.Type ==
                                WM_AT_RSP_PRE_EMBEDDED_TREATMENT)
                        {
                                /* Filtering Embedded processing */
                                ...
                        }
                        else if (Message->Body.ATRspPreParser.Type ==
                                                WM_AT_RSP_PRE_BRAODCAST)   {
                                /* Spying Embedded processing */
                                ...
                        }
                ...
        }
        return OK;
}
```

3.3.6 The wm_atSendRspExternalApp Function

The wm_atSendRspExternalApp function sends an AT response to the External Application, in case of AT command pre-parsing.
Its prototype is:

```
void wm_atSendRspExternalApp
(                                    u16              AtStringSize,
                                     ascii            *AtString );
```

3.3.6.1 Parameters

AtString:

Any AT response string in ASCII characters (terminated by a 0x00 character). This string is sent on the serial link without any change : it should include “\r\n” characters at the end and/or the beginning of the string.

AtStringSize:

Size of the previous *AtString* parameter. It equals the length + 1 and includes the 0x00 character.

3.3.6.2 Required Header

Wm_at.h

3.3.6.3 Notes

❑ This function should be used to transmit to the external application the responses received by the embedded application through the WM_AT_RESPONSE message.

3.3.7 The wm_atSendUnsolicitedExternalApp Function

The wm_atSendUnsolicitedExternalApp function sends an AT unsolicited response to the External Application.
Its prototype is:

```
void wm_atSendUnsolicitedExternalApp (  u16              AtStringSize,
                                        ascii            *AtString );
```

3.3.7.1 Parameters

AtString:

Any AT unsolicited response string in ASCII characters (terminated by a 0x00 character). This string is sent on the serial link without any change : it should include “\r\n” characters at the end and/or the beginning of the string.

AtStringSize:

Size of the previous AtString parameter. It equals the length + 1 and includes the 0x00 character.

3.3.7.2 Required Header

Wm_at.h

3.3.7.3 Notes

- ❑ An unsolicited response string sent by the wm_atSendUnsolicitedExternalApp function will only be displayed on the serial link when the Wavecom AT task is not busy by a command processing. If it is busy in a such processing, the unsolicited response string is stored, and displayed at the end of the process (after the terminal AT response).
- ❑ Sending an AT response by the wm_atSendRspExternalApp function will display all previously stored unsolicited responses (after this response display).
- ❑ This function should be used to transmit to the external application the unsolicited responses received by the embedded application through the WM_AT_UNSOLICITED message.

3.3.8 The wm_atSendIntermediateExternalApp Function

The wm_atSendIntermediateExternalApp function sends an AT intermediate response to the External Application.

Its prototype is:

```
void wm_atSendIntermediateExternalApp ( u16        AtStringSize,
                                        ascii      *AtString );
```

3.3.8.1 Parameters

AtString:

Any AT intermediate response string in ASCII characters (terminated by a 0x00 character). This string is sent on the serial link without any change : it should include “\r\n” characters at the end and/or the beginning of the string.

AtStringSize:

Size of the previous AtString parameter. It equals the length + 1 and includes the 0x00 character.

3.3.8.2 Required Header

Wm_at.h

3.3.8.3 Notes

- ❑ An intermediate response string sent by the wm_atSendIntermediateExternalApp function will always display this string on the serial link, either the Wavecom AT task is busy on a command processing or not.
- ❑ Previously stored unsolicited responses will not be displayed after a call to the wm_ atSendIntermediateExternalApp function.
- ❑ This function should be used to transmit to the external application the intermediate responses received by the embedded application through the WM_AT_INTERMEDIATE message.

3.4 OS API

3.4.1 The wm_osStartTimer Function

The wm_osStartTimer function sets up a timer associated to an existing *TimerId*.
Its prototype is:

```
s32    wm_osStartTimer       ( u8              TimerId,
                               bool            bCyclic,
                               u32             TimerValue );
```

3.4.1.1 Parameters

TimerId:

Timer identifier: the range 0 to WM_OS_MAX_TIMER_ID is accepted.

BCyclic:

This parameter may have one of the following values:

❑ **TRUE**: the timer is cyclic and is automatically set up when a cycle is over,

❑ **FALSE**: in case the timer has only one cycle.

TimerValue:

Timer unit :100 ms.

3.4.1.2 Return Values

The return parameter is positive or null if the timer is set up and negative if not.

3.4.1.3 Required Header

```
wm_os.h
```

3.4.1.4 Note

❑ The timer expiry indication is received by the Embedded Application through a message. This message is available as a parameter of the **wm_apmAppliParser()** function with the *MsgTyp* parameter set to WM_OS_TIMER (see § 3.2.3: “The wm_apmAppliParser”).

3.4.1.5 Example: Managing a Timer

The range 0 to WM_OS_MAX_TIMER_ID is accepted. A timer may or may not be cyclic. An example of setting up a timer is given below:

```
/* Timer start, not cyclic, value = 1second */
wm_osStartTimer( 1, FALSE, 10 );
```

An example of receiving a timer expiry event is given below:

```
s32 wm_apmAppliParser (wm_apmMsg_t * Message)
{
        ascii *     strBuffer;
        u16         nLenBuffer;

        switch (Message->MsgTyp)
        {
                ....
                case WM_OS_TIMER:

                ...
        }
        return OK;
}
```

3.4.2 The wm_osStopTimer Function

The wm_osStopTimer function stops the timer identified by TimerId.
Its prototype is:

s32 wm_osStopTimer (u8 *TimerId*);

3.4.2.1 Parameter

TimerId:

Timer identifier: the range 0 to WM_OS_MAX_TIMER_ID is accepted.

3.4.2.2 Return Values

The return parameter is the remaining time if the timer was still running, and a negative value otherwise.

3.4.2.3 Required Header

wm_os.h

3.4.3 The wm_osDebugTrace Function

The wm_osDebugTrace function is aimed at trace managing.
Its prototype is:

```
s32 wm_osDebugTrace ( u8 Level, char *Format, ... );
```

3.4.3.1 Parameters

Level:

Used to differentiate the traces. The PC trace software gives access to level configuration.

Format:

Used to specify a string and the corresponding formats (like the printf function), as far as the data to trace is concerned. The supported formats are ‘c’, ’x’, ‘X’, ‘u’, ‘d’.
Up to 6 parameters may be included in the *Format* string.
As the ‘s’ format is not supported, the way to display a char * string is to replace the Format string by this char, without any parameters.

…:

Represents the list of data to be traced.

3.4.3.2 Required Header

```
wm_os.h
```

3.4.3.3 Returned values

A positive or null value indicates that the trace has been sent; otherwise a negative error value is sent.

3.4.3.4 Example: Inserting Debug Information

Debug information is included in the Embedded Application, and therefore it uses ROM space and CPU resources.
The Target Monitoring Tool is used to display the Debug information.
An example of tracing an informational message is given below:

```
wm_osDebugTrace ( 1, “This is an informational message on level 1” );
/* To visualise this, the Target Monitoring Tool must be configured to extract level 1
traces */
/* The result string using the Target Monitoring Tool should be:
 “This is an informational message on level 1” */
```

An example of tracing an informational message using a decimal parameter is given below:

```
u8 param =12;
wm_osDebugTrace ( 2, “This is an informational message on level 2 with 1 parameter
=%d”, param );
/* To visualise this, the Target Monitoring Tool must be configured   to extract level 2
traces */

/* The result string using the Target Monitoring Tool should be:
 “This is an informational message on level 2 with 1 parameter =12” */
```

An example of tracing a string is given below:

```
ascii String[]="Hello World";
wm_osDebugTrace ( 3, String );
/* To visualise this, the Target Monitoring Tool must be configured to extract level 3
traces */
/* The result string on Target Monitoring Tool should be:
 "Hello World" */
```

3.4.4 The wm_osDebugFatalError Function

The wm_osDebugFatalError function is the fatal error function: it stores the error code and then performs a reboot.

Its prototype is:

```
s32  wm_osDebugFatalError ( char * Message );
```

3.4.4.1 Parameters

Message:

String to be displayed whenever an error occurs.

3.4.4.2 Required Header

```
wm_os.h
```

3.4.4.3 Returned Value

A negative error value indicates that the fatal error did not happened.

3.4.4.4 Note

The reboot is performed after the call to the fatal error function. In order to ensure the downloading of a new binary file after a fatal error has been detected, the User software startup is delayed 20 sec.

Therefore, in order not to miss any event, the application has to handle a startup delay of 20 sec.

3.4.5 Important Note on Data Flash Management

The Data Flash Identifiers are organized in the memory as follows:

❑ a 10-byte header,

❑ the body.

An application cannot use more than 5KB of Data Flash. Therefore, depending on the size of the stored data, the number of available Identifiers will vary.

For instance:

❑ if the application needs to store 1 byte of data, the number of available Identifiers is equal to 5000/11 = 454 Identifiers.

❑ if the application needs to store 100 bytes of data, the number of available Identifiers is equal to 5000/110 = 45 Identifiers.

ATTENTION :

The identifiers are represented by a **u16** value. Any value can be used as identifier, except **0xFFFF**.

3.4.6 The wm_osWriteFlashData Function

The wm_osWriteFlashData function is used to write data into Flash ROM. The corresponding identifier is assigned to the stored data.
The prototype of this function is:

```
s32 wm_osWriteFlashData ( u16 Id, u16 DataLen, u8 *Data );
```

3.4.6.1 Parameters

Id:
Identifier assigned to the stored data.

DataLen:
Length of the data to be stored (in bytes).

Data:
Pointer to the data to be stored.

3.4.6.2 Return Values

The return parameter is positive or null if data has been written, and negative if not.

3.4.6.3 Required Header

```
wm_os.h
```

3.4.7 The wm_osReadFlashData Function

The wm_osReadFlashData function is used to read data identified by Id from the Flash ROM.
Its prototype is:

```
s32 wm_osReadFlashData ( u16 Id, u16 DataLen, u8 *Data );
```

3.4.7.1 Parameters

Id:
Identifier assigned to the stored data.

DataLen:
Length of the data to be read (in bytes).

Data:
Pointer to the data to be read.

3.4.7.2 Return Values

The return parameter is the length to be read and copied to **Data* on success, and a negative value on error.

3.4.7.3 Required Header

```
wm_os.h
```

3.4.8 The wm_osGetLenFlashData Function

The wm_osGetLenFlashData function supplies the length of the data stored in Flash ROM and identified by Id.
Its prototype is:

```
s32 wm_osGetLenFlashData ( u16 Id );
```

3.4.8.1 Parameter

Id:
Identifier assigned to the stored data.

3.4.8.2 Return Values

The return parameter is the byte length of the data identified by Id. If it is negative, an error has occurred.

3.4.8.3 Required Header

```
wm_os.h
```

3.4.9 The wm_osDeleteFlashData Function

The wm_osDeleteFlashData function deletes the data stored in Flash ROM and identified by Id.
Its prototype is:

```
s32 wm_osDeleteFlashData ( u16 Id );
```

3.4.9.1 Parameter

Id:
Identifier assigned to the stored data.

3.4.9.2 Return Values

The return parameter is positive or null if the data have been deleted, and negative if not.

3.4.9.3 Required Header

```
wm_os.h
```

3.4.10 The wm_osGetAllocatedMemoryFlashData Function

The wm_osGetAllocatedMemoryFlashData function returns the quantity of allocated memory in Flash ROM.
Its prototype is:

```
s32 wm_osGetAllocatedMemoryFlashData ( void );
```

3.4.10.1 Return Values

The return parameter is the quantity of allocated memory in Flash ROM (Unit : bytes) on success, and a negative value on error.

3.4.10.2 Required Header

```
wm_os.h
```

3.4.11 The wm_osGetFreeMemoryFlashData Function

The wm_osGetFreeMemoryFlashData function returns the quantity of available memory in Flash ROM.

Its prototype is:

```
s32  wm_osGetFreeMemoryFlashData ( void );
```

3.4.11.1 Return values***

The return parameter is the quantity of free memory in Flash ROM on success, and a negative value on error.

3.4.11.2 Required Header

```
wm_os.h
```

3.4.12 The wm_osDeleteAllFlashData Function

The wm_osDeleteAllFlashData function deletes all the data previously stored in flash memory by the embedded application.

Its prototype is :

```
s32  wm_osDeleteAllFlashData ( void );
```

3.4.12.1 Return values

If the delete operation if successful, returns the number of deleted objects.

Otherwise, returns a negative error value.

3.4.12.2 Required Header

```
wm_os.h
```

3.4.13 Example: Managing Data Flash Objects

5KB of Data Flash objects are available for Embedded Applications.

Data Flash objects are organized in Ids and managed by the Embedded Application.

An Example related to Data Flash reading/writing is given below:

```
s32 LengthRead;
s32 Length;
u8* ptr;
u16 Id;
s32 Writen;
FlashId = 112;
/* Get the len */
Length = wm_osGetLenFlashData (FlashId);
Ptr = wm_osGetHeapMemory (Length);
/* Read the Flash Id item */
LengthRead  = wm_osReadFlashData  (FlashId, Length, Ptr);
Ptr[3] = 0x10; /* Change something */
/* Write the modified Flash Id item */
Writen = wm_osWriteFlashData  (FlashId, Length, Ptr);
```

3.4.14 The wm_osGetHeapMemory Function

The wm_osGetHeapMemory function gets memory from the Embedded heap.
Its prototype is:

```
void *wm_osGetHeapMemory ( u16 MemorySize);
```

3.4.14.1 Parameter

MemorySize:

Requested size.

3.4.14.2 Return Values

The return parameter is the the memory address or is NULL if an error has occurred.

3.4.14.3 Required Header

```
wm_os.h
```

3.4.15 The wm_osReleaseHeapMemory Function

The wm_osReleaseHeapMemory function releases the previously reserved memory.
Its prototype is:

```
s32 wm_osReleaseHeapMemory (void * ptrData );
```

3.4.15.1 Parameter

PtrData:

Points to the reserved memory.

3.4.15.2 Return Values

The return parameter is positive or null if the reserved memory has been released, and negative if not.

3.4.15.3 Required Header

```
wm_os.h
```

3.4.16 Example: RAM management

32 KB of RAM are available for Embedded Applications and the provided Wavecom library manages this RAM.
An example of the RAM request function is given below:

```
void *ptr;
ptr = wm_osGetHeapMemory ( 1000 ); /* 1000 bytes are asked */
```

An example of the RAM release function is given below:

```
wm_osReleaseHeapMemory (ptr);
```

3.5 Flow Control Manager API

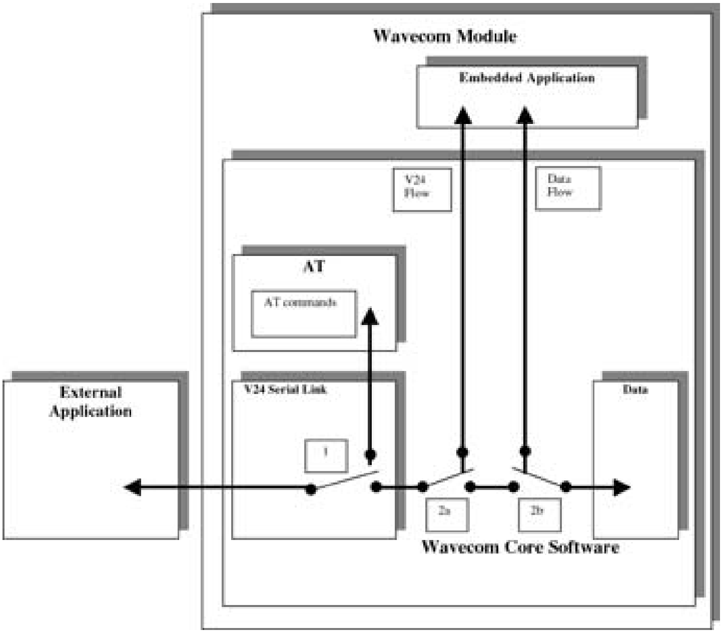


Figure 2: Flow Control Function

The Flow Control Manager API provides two IO flows to the embedded application: one from the V24 serial link, and one from a Data Communication (though the GSM air interface).

By default, these flows are closed (in Figure 2, Switches 2a and 2b are closed to transmit all data directly between the V24 serial link and Data communication).

The embedded application can use the **wm_fcmOpenDataAndV24()** (see § 3.5.1: "The wm_fcmOpenDataAndV24") and **wm_fcmCloseDataAndV24()** (see § 3.5.2: "The wm_fcmCloseDataAndV24") functions to open or close these flows.

One flow cannot be opened alone (on Figure 2, the switches 2a and 2b are always closed or opened together).

The Switch 1 function is described in § 3.6.1: "The wm_ioSerialSwitchState."

3.5.1 The wm_fcmOpenDataAndV24 Function

The wm_fcmOpenDataAndV24 function opens two flows between the embedded application and the V24 serial link, and between the application and a Data communication.

Its prototype is:

```
s32  wm_fcmOpenDataAndV24 (     u16          DataMaxToReceiveFromData,
                                u16          DataMaxToReceiveFromV24 );
```

3.5.1.1 Parameters

DataMaxToReceiveFromData:

Maximum block size to be sent to the embedded application from a Data communication. This size can not exceed **270 bytes**.

DataMaxToReceiveFromV24:

Maximum block size to be sent to the embedded application from the V24 serial link. This size can not exceed **120 bytes**.

3.5.1.2 Required Header

```
Wm_fcm.h
```

3.5.1.3 Return Value

The returned value is not relevant.

3.5.1.4 Notes

- ❑ The flow opening response is received by the Embedded Application through a message. This message is available as a parameter of the **wm_apmAppliParser()** function with the *MsgTyp* parameter set to WM_FCM_OPEN_FLOW (see § 3.2.3: “The wm_apmAppliParser”). The embedded application will receive a message for each type of flow (V24 serial link and Data).
- ❑ The DataMaxToSend parameter of the WM_FCM_OPEN_FLOW message informs the embedded application of the maximum data block size it can send on this flow. If this parameter is 0, there is no size limitation.
- ❑ The **wm_fcmOpenDataAndV24()** function **must** be called **before** using the “**ATD**” command to set up a data call.

3.5.2 The wm_fcmCloseDataAndV24 Function

The wm_fcmCloseDataAndV24 function closes the two flows between the embedded application and V24 serial link, and between the application and a Data communication.
Its prototype is:

```
s32 wm_fcmCloseDataAndV24 ( void );
```

3.5.2.1 Required Header

```
Wm_fcm.h
```

3.5.2.2 Return Value

The returned value is not relevant.

3.5.2.3 Notes

- ❑ The flow closing response is received by the Embedded Application through a message. This message is available as a parameter of the **wm_apmAppliParser()** function with the *MsgTyp* parameter set to WM_FCM_CLOSE_FLOW (see § 3.2.3: "The wm_apmAppliParser"). The embedded application will receive a message for each flow type (V24 serial link and Data).
- ❑ The **wm_fcmCloseDataAndV24()** function **must** be called **after** any data call release.

3.5.3 The wm_fcmSubmitData Function

The wm_fcmSubmitData function submits a data block to the Flow Control Manager.
Its prototype is:

```
s32 wm_fcmSubmitData (          wm_fcmFlow_e            Flow,
                                wm_fcmSendBlock_t *     fcmDataBlock );
```

3.5.3.1 Parameters

Flow:

Specifies the IO flow where the data are sent; the possible values are:

```
typedef enum                {
        WM_FCM_DATA,
        WM_FCM_V24
} wm_fcmFlow_e;
```

WM_FCM_DATA represents the data flow of a Data Communication.
WM_FCM_V24 represents the data flow of the V24 serial link.

fcmDataBlock:

Pointer on a wm_fcmSendBlock_t structure, allocated (see § 3.4.13: "The wm_osGetHeapMemory ") and filled by the embedded application before sending.
The definition of this structure is as follows:

```
typedef struct              {
        u16 Reserved1[4];
        u16 DataLength;             /* number of byte of data to send */
        u16 Reserved2[5];
        u8  Data[1];                /* data to send */
} wm_fcmSendBlock_t;
```

3.5.3.2 Returned Values

WM_FCM_OK means the data block is sent, the memory allocated for fcmDataBlock is released, and the embedded application may go on sending more data blocks.
WM_FCM_EOK_NO_CREDIT means the data block is sent and the memory allocated for fcmDataBlock is released, but the embedded application must wait for the **WM_FCM_RESUME_DATA_FLOW** message before sending more data blocks. This message is available as a parameter of the **wm_apmAppliParser()** function (see § 3.2.3: "The wm_apmAppliParser").
WM_FCM_ERR_NO_CREDIT means the data block is not sent and the memory allocated for fcmDataBlock is not released. The embedded application must wait for the **WM_FCM_RESUME_DATA_FLOW** message before sending more data blocks. This message is available as a parameter of the **wm_apmAppliParser()** function (see § 3.2.3: "The wm_apmAppliParser").
WM_FCM_ERR_NO_LINK means the flow is not opened. The data block is not sent and the memory allocated for fcmDataBlock is not released.
WM_FCM_ERR_UNKNOWN_FLOW means the embedded application used an incorrect flow ID. The data block is not sent and the memory allocated for fcmDataBlock is not released.

3.5.3.3 Required Header

Wm_fcm.h

3.5.3.4 Notes

❑ A successful data send by the **wm_fcmSubmitData()** function (with WM_FCM_OK ot WM_FCM_EOK_NO_CREDIT return code) will result in the receipt of a WM_OS_RELEASE_MEMORY message by the Embedded Application. This message is available as a parameter of the **wm_apmAppliParser()** function with the *MsgTyp* parameter set to WM_OS_RELEASE_MEMORY (see § 3.2.3: "The wm_apmAppliParser").

❑ You should not call the **wm_fcmSubmitData()** function more than once in the same message treatment. The embedded application should set a timer between each data block sending on the IO flows.

❑ Set a timer between the last data block sending on an IO flow, and this flow closing operation. Also, a timer should be set between the last data block sending on the V24 flow, and a call to the **wm_ioSwitchSerialState** (**WM_IO_SERIAL_AT_MODE**) function.

❑ In remote task mode, as the serial link is strongly used (AT commands and responses, traces and messages between the remote task and the target software), a data send operation on the V24 flow with high speed rate will not work. The embedded application should send data blocks on the V24 flow a very low speed rate, in remote task mode.

3.5.4 Receive Data Blocks

The embedded application may receive data blocks from an opened Data or V24 IO flow, through the WM_FCM_RECEIVE_BLOCK message. This message is available as a parameter of the **wm_apmAppliParser()** function (see § 3.2.3: "The wm_apmAppliParser").

3.5.4.1 Message Parameters

This is the WM_FCM_RECEIVE_BLOCK message structure:

```
typedef struct          {
        u16                                 DataLength;       /* number of bytes received */
        u8                                  Reserved1[2];
        wm_fcmFlow_e                        FlowId;           /* IO flow ID */
        u8                                  Reserved2[7];
        u8                                  Data[1];          /* data received */
} wm_fcmReceiveBlock_t;
```

DataLength:

Number of data bytes received in Data parameter from this flow. This size will not exceed DataMaxToReceiveFromData or DataMaxToReceiveFromV24 parameters (depending on the flow type) of the **wm_fcmOpenDataAndV24()** function
(see § 3.5.1: "The wm_fcmOpenDataAndV24").

FlowID:

Specifies the opened IO flow from where the data are received. The possible values are:

```
typedef enum                    {
        WM_FCM_DATA,
        WM_FCM_V24
} wm_fcmFlow_e;
```

WM_FCM_DATA represents the data flow of a Data Communication.
WM_FCM_V24 represents the data flow of the V24 serial link.

Data:

Data block received from the IO flow. The memory allocated for Data parameter will be released at the end of the **wm_apmAppliParser()** function
(see § 3.2.3: "The wm_apmAppliParser").

3.5.4.2 Required Header

```
Wm_fcm.h
```

3.5.4.3 Notes

- ❑ When the embedded application has treated one or more data blocks, it should inform the Flow Control Manager to release credits, in order to receive more data, by using the **wm_fcmCreditToRelease()** function
(see § 3.5.5: "The wm_fcmCreditToRelease").

3.5.5 The wm_fcmCreditToRelease Function

The wm_fcmCreditToRelease function informs the Flow Control Manager that the embedded application has treated some data blocks, and is ready to receive more data. This credit release system provides more security for the data transfer.
Its prototype is:

```
s32  wm_fcmCreditToRelease (  wm_fcmFlow_e        Flow,
                              u8                  Credits );
```

3.5.5.1 Parameters

Flow:

Specifies the IO flow on which the Flow Control Manager may release credits.
The possible values are:

```
typedef enum                 {
        WM_FCM_DATA,
        WM_FCM_V24
} wm_fcmFlow_e;
```

WM_FCM_DATA represents the data flow of a data communication.
WM_FCM_V24 represents the data flow of the V24 serial link.

Credits:

Specifies the number of credits the embedded application wants the Flow Control Manager to release. This represents the number of data blocks received and treated by the embedded application.
For example: when the embedded application has received and treated 3 data blocks (i.e. 3 WM_FCM_RECEIVE_BLOCK messages), it should inform the Flow Control Manager by calling the **wm_fcmCreditToRelease()** function with the Credits parameter set to 3.

3.5.5.2 Returned Values

The returned value is ≥ 0 if the credits are released, otherwise it is negative (an error occurred and the credits are not released).

3.5.5.3 Required Header

```
Wm_fcm.h
```

3.6 Input Output API

This API manages Serial Link State and Gpio operations.

3.6.1 Serial Link State functions

3.6.1.1 The wm_ioSerialSwitchState Function

The wm_ioSerialSwitchState function sets the serial link mode: AT command computing, or direct data transmission through the V24 Serial Link Flow.
Its prototype is:

```
void  wm_ioSerialSwitchState ( wm_ioSerialSwitchState_e SerialState );
```

3.6.1.1.1 Parameters

SerialState:

Specifies the requested state of the Serial Link. The possible values are defined bellow:

```
typedef enum                         {
        WM_IO_SERIAL_AT_MODE,
        WM_IO_SERIAL_DATA_MODE,
        WM_IO_SERIAL_ATO
} wm_ioSerialSwitchState_e;
```

WM_IO_SERIAL_AT_MODE represents the AT commands computing mode. In this mode, data received from V24 serial link are parsed and treated like AT commands.
WM_IO_SERIAL_DATA_MODE represents the direct data transmission mode. In this mode, data received from V24 serial link are transmitted without treatment through the V24 Serial Link Flow.
WM_IO_SERIAL_ATO is used only if the external application sent a "+++" string, in order to switch the V24 interface in "ONLINE" mode (see "Notes").

3.6.1.1.2 Required Header

```
Wm_io.h
```

3.6.1.1.3 Notes

- ❑ The serial mode switching response is received by the Embedded Application through a message. This message is available as a parameter of the **wm_apmAppliParser()** function with the *MsgTyp* parameter set to WM_IO_SERIAL_SWITCH_STATE_RSP (see § 3.2.3: "The wm_apmAppliParser"). The SerialMode parameter of this message is the requested Serial Link Mode; if the RequestReturn parameter is negative, an error occurred, and the Serial Link Mode does not change.
- ❑ The **wm_ioSerialSwitchState()** function is not allowed if the V24 Serial Link and the Data Flows are not opened by the embedded application (see § 3.5.1: "The wm_fcmOpenDataAndV24").
  In this case, the WM_IO_SERIAL_SWITCH_STATE_RSP message will always return a negative RequestReturn parameter.
- ❑ In Figure 2 (see § 3.5: "Flow Control Manager API"), the **wm_ioSerialSwitchState()** function controls Switch 1.

IMPORTANT NOTES

❑ Using the **ATD** command to begin a data call (from external or embedded application) will switch the serial link to WM_IO_SERIAL_DATA_MODE state after the **CONNECT** response.

❑ When a data call is released (from the remote party, or with the **ATH** command), the serial link is switched to WM_IO_SERIAL_AT_MODE state (respectively after the **NO CARRIER** or **OK** response).

❑ Sending the "+++" sequence from an external application while the serial link is in WM_IO_SERIAL_DATA_MODE state will switch it to WM_IO_SERIAL_AT_MODE state after the **OK** response, during or out of a data call. The "+++" sequence must be preceded and followed by a period of one second without character sending; otherwise the serial link state will not switch to WM_IO_SERIAL_AT_MODE.

❑ During a data call, the **ATO** command will switch the serial link to WM_IO_SERIAL_DATA_MODE state after the **OK** response.

❑ Out of data call, the **ATO** command is not allowed; the embedded application may use the WM_IO_SERIAL_ATO mode to return to the WM_IO_SERIAL_DATA_MODE state.

3.6.2 Gpio types and functions

3.6.2.1 Types

3.6.2.1.1 The wm_ioConfig_t structure

This structure is used by the wm_ioAllocate function in order to set the reserved Gpio parameters.

```
typedef struct
{
        wm_ioLabel_u                  eLabel;
        u32                           Pad;
        wm_ioDirection_e              eDirection;
        wm_ioState_e                  eState;
} wm_ioConfig_t;
```

The **eLabel** member represents the Gpio label.
The **eDirection** member represents the Gpio direction.
The **eState** member represents the Gpio state.

3.6.2.1.2 The wm_ioLabel_u union

This union represents the different Gpio labels, depending on the used product.

```
typedef union
{
        wm_ioLabel_Quik_e             Quik_Label;
        wm_ioLabel_Pac_e              Pac_Label;
} wm_ioLabel_u;
```

The **Quik_Label** member must be used on Wismo Quik based products.
The **Pac_Label** member must be used on Wismo Pac based products.

3.6.2.1.2.1 Wismo Quik Gpio Labels

The Gpio labels for Wismo Quik based products are defined by the values below :

```
typedef enum
{

        WM_IO_QUIK_GPI              = 0x00000001,

        WM_IO_QUIK_GPO_1            = 0x00000004,
        WM_IO_QUIK_GPO_2            = 0x00000008,

        WM_IO_QUIK_GPIO_0           = 0x00000010,
        WM_IO_QUIK_GPIO_4           = 0x00000100,
        WM_IO_QUIK_GPIO_5           = 0x00000200

} wm_ioLabel_Quik_e;
```

3.6.2.1.2.2 Wismo Pac Gpio Labels

The Gpio labels for Wismo Pac based products are defined by the values below:

```
typedef enum
{

        WM_IO_PAC_GPI               = 0x00000001,

        WM_IO_PAC_GPIO_0            = 0x00000008,
        WM_IO_PAC_GPIO_2            = 0x00000020,
        WM_IO_PAC_GPIO_3            = 0x00000040,
        WM_IO_PAC_GPIO_4            = 0x00000080,
        WM_IO_PAC_GPIO_5            = 0x00000100

} wm_ioLabel_Pac_e;
```

3.6.2.1.3 The wm_ioDirection_e type

This type represents the direction used for a Gpio.

```
typedef enum
{
        WM_IO_OUTPUT,
        WM_IO_INPUT,
        WM_IO_NORMAL
} wm_ioDirection_e;
```

The **WM_IO_OUTPUT** constant is used to set a Gpio as an output.
The **WM_IO_INPUT** constant is used to set a Gpio as an input.

A GPI must always be allocated with the WM_IO_INPUT direction.
A GPO must always be allocated with the WM_IO_NORMAL direction.

3.6.2.1.4 The wm_ioState_e type

This type represents the state of a Gpio.

```
typedef enum
{
        WM_IO_LOW,
        WM_IO_HIGH
} wm_ioState_e;
```

The **WM_IO_LOW** constant represents the low state of a Gpio.
The **WM_IO_HIGH** constant represents the high state of a Gpio.

3.6.2.1.5 The wm_ioSetDirection_t structure

This type is used by the wm_ioSetDirection function to set a Gpio to a new direction.

```
typedef struct
{
        wm_ioLabel_u                    eLabel;
        wm_ioDirection_e                eDirection;
} wm_ioSetDirection_t;
```

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

3.6.2.1.6 Return values definition

WM_IO_PROC_DONE (0) : the function processing is done successfuly.
WM_IO_UNKNOWN_TYPE (-1) : a direction parameter has an incorrect value.
WM_IO_INPUT_CANT_BE_SET (-2) : the function tried to set an Input pin.
WM_IO_OUTPUT_CANT_BE_READ (-3) : the function tried to read an Output pin.
WM_IO_NO_MORE_HANDLES_LEFT (-4) : no more handle to allocate the requested Gpios.
WM_IO_EXCEED_MAX_NUMBER (-5) : a parameter exceed the allowed range value.
WM_IO_UNALLOCATED_HANDLE (-6) : a handle parameter has an incorrect value.
WM_IO_INCOHERENCE_BETWEEN_HANDLE_AND_MASK (-7) : the function tried to use a Gpio mask with an incorrect Handle.
WM_IO_INCOHERENCE_BETWEEN_DIRECTION_AND_MASK (-8) : the function tried to set an input pin direction to output, or an output pin direction to input.
WM_IO_IO_ALREADY_USED (-9) : the function tried to allocate a Gpio already allocated on another Handle.
WM_IO_INCOHERENCE_BETWEEN_HANDLE_AND_IO_NUMBER (-18) : the function tried to use a Gpio value with an incorrect Handle.

3.6.2.2 The wm_ioAllocate Function

The wm_ioAllocate function reserves one or more Gpio(s) for the embedded application use.

Its prototype is:

```
s32 wm_ioAllocate (                 u32 NbGpioToAllocate,
                                    wm_ioConfig_t * GpioCustomerConfig );
```

3.6.2.2.1 Parameters

NbGpioToAllocate:
Size of the GpioCustomerConfig array.

GpioCustomerConfig:
Array of values, defined by the wm_ioConfig_t structure (see §3.6.2.1.1).

For each member of this array:

- ❑ eLabel represents the label of the requested Gpio, Gpi or Gpo, depending on the used product.
- ❑ eDirection represents the direction used for this Gpio.
- ❑ eState represents the state of the requested Gpio.

3.6.2.2.2 Returned Values
If the Gpio allocate operation is successful, the returned value is a positive or null Handle, which must be used in all further operations on the reserved Gpios.

Otherwise, a negative returned value represents an error (cf § 3.6.1.2.6 "Returned values definition").

3.6.2.2.3 Required Header

```
Wm_io.h
```

3.6.2.2.4 Notes

- ❑ The eDirection member of the wm_ioConfig_t structure is only significant for Gpio pins. Gpi pins should be always set as an input ; Gpo pins should be always set as an output. Otherwise, the eDirection parameter is not taken into account.
- ❑ The eState member of the wm_ioConfig_t structure is only significant for pins set as an output by the eDirection parameter. Otherwise, the eState parameter is not taken into account.
- ❑ After a successful allocation, Gpio allocated by the embedded application are no more available for AT commands (AT+WIOR, AT+WIOW, AT+WIOM).

3.6.2.3 The wm_ioRelease Function

The **wm_ioRelease** function allows to release one or more Gpio reserved by the **wm_ioAllocate** function.
Its prototype is:

```
s32 wm_ioRelease (                 s32 Handle,
                                   u32 NbGpioToRelease,
                                   wm_ioLabel_u * GpioCustomerLabel );
```

3.6.2.3.1 Parameters

Handle:

Handle returned by the wm_ioAllocate function. All Gpios of GpioCustomerLabel parameter must be related to this Handle.

NbGpioToRelease:

Size of the GpioCustomerLabel array.

GpioCustomerLabel:

Array of values, defined by the **wm_ioLabel_u** union (see §3.6.2.1.2).

Each member of this array represents the label of one Gpio to release.

3.6.2.3.2 Returned Values

0 : successful completion

Otherwise, a negative returned value represents an error (cf § 3.6.1.2.6 "Returned values definition").

3.6.2.3.3 Required Header

```
Wm_io.h
```

3.6.2.3.4 Notes

- ❑ If one of the given Gpio labels is not related to the given Handle, the wm_ioRelease function will fail.
- ❑ After a successful release, Gpio released control is resumed by AT commands (AT+WIOR, AT+WIOW, AT+WIOM).

3.6.2.4 The wm_ioSetDirection Function

The wm_ioSetDirection function allows to change the direction of an allocated Gpio. Its prototype is:

```
s32 wm_ioSetDirection (          s32 Handle,
                                 u32 NbGpioToChangeDir,
                                 wm_ioSetDirection_t * GpioDirection );
```

3.6.2.4.1 Parameters

Handle:

Handle returned by the wm_ioAllocate function. All Gpios of GpioDirection parameter must be related to this Handle.

NbGpioToChangeDir:

Size of the GpioDirection array.

GpioDirection:

Array of values, defined by the wm_ioSetDirection_t structure (see §3.6.2.1.5).

For each member of this array:

- ❑ **eLabel** represents the label of the Gpio, Gpi or Gpo to change direction, depending on the used product.
- ❑ **eDirection** represents the new direction to use for this Gpio.

3.6.2.4.2 Returned Values

0 : successful completion

Otherwise, a negative returned value represents an error (cf § 3.6.1.2.6 "Returned values definition").

3.6.2.4.3 Required Header

Wm_io.h

3.6.2.4.4 Notes

- ❑ If one of the given Gpio labels is not related to the given Handle, the wm_ioSetDirection function will fail.
- ❑ This function is only useful for Gpio pins. Gpi or Gpo pins direction should not be changed.

3.6.2.5 The wm_ioRead Function

The wm_ioRead function allows to read the current state of one or more allocated Gpio(s). Its prototype is :

```
s32 wm_ioRead (                    s32 Handle,
                                   u32 Gpio,
                                   u32 * GpioState );
```

3.6.2.5.1 Parameters

Handle:

Handle returned by the wm_ioAllocate function. All Gpios of Gpio parameter must be related to this Handle.

Gpio:

Mask designating the Gpio(s) to read. This mask is obtained by performing a OR with members of the wm_ioLabel_u union.

GpioState:

Mask used to return the read states. Each bit of this mask represents the state of the corresponding Gpio in the "Gpio" parameter.

3.6.2.5.2 Returned Values

0 : successful completion

Otherwise, a negative returned value represents an error (cf § 3.6.1.2.6 "Returned values definition").

3.6.2.5.3 Required Header

Wm_io.h

3.6.2.5.4 Notes

- ❑ If one of the given Gpio labels is not related to the given Handle, the wm_ioRead function will fail.

3.6.2.6 The wm_ioSingleRead Function

The wm_ioSingleRead function allows to read the current state of one single allocated Gpio.
Its prototype is:

```
s32 wm_ioSingleRead (            s32 Handle,
                                 u32 Gpio );
```

3.6.2.6.1 Parameters

Handle:
Handle returned by the wm_ioAllocate function. The Gpio parameter must be related to this Handle.

Gpio:
Value designating the Gpio to read, member of the wm_ioLabel_u union.

3.6.2.6.2 Returned Values

If the read operation is successful, the function returns the Gpio state.

Otherwise, a negative returned value represents an error (cf § 3.6.1.2.6 “Returned values definition”).

3.6.2.6.3 Required Header

```
Wm_io.h
```

3.6.2.6.4 Notes

❑ If the given Gpio label is not related to the given Handle, the wm_ioSingleRead function will fail.

3.6.2.7 The wm_ioWrite Function

The wm_ioWrite function allows to define a new state for one or more allocated Gpio(s).
Its prototype is :

```
s32 wm_ioWrite (                 s32 Handle,
                                 u32 Gpio,
                                 u32 GpioState );
```

3.6.2.7.1 Parameters

Handle:
Handle returned by the wm_ioAllocate function. All Gpios of Gpio parameter must be related to this Handle.

Gpio:
Mask designating the Gpio(s) to write. This mask is obtained by performing a OR with members of the wm_ioLabel_u union.

GpioState:
Mask used to indicate the different states to write. Each bit of this mask represents the state of the corresponding Gpio in the “Gpio” parameter.

3.6.2.7.2 Returned Values

0 : successful completion

Otherwise, a negative returned value represents an error (cf § 3.6.1.2.6 "Returned values definition").

3.6.2.7.3 Required Header

```
Wm_io.h
```

3.6.2.7.4 Notes

- ❑ If one of the given Gpio labels is not related to the given Handle, the wm_ioWrite function will fail.

3.6.2.8 The wm_ioSingleWrite Function

The wm_ioSingleWrite function allows to define a new state for one single allocated Gpio. Its prototype is:

```
s32 wm_ioSingleWrite (          s32 Handle,
                                u32 Gpio
                                u32 State );
```

3.6.2.8.1 Parameters

Handle:

Handle returned by the wm_ioAllocate function. The Gpio parameter must be related to this Handle.

Gpio:

Value designating the Gpio to write, member of the wm_ioLabel_u union.

State:

Value designating the State to write (High or Low).

3.6.2.8.2 Returned Values

0 : successful completion

Otherwise, a negative returned value represents an error (cf § 3.6.1.2.6 "Returned values definition").

3.6.2.8.3 Required Header

```
Wm_io.h
```

3.6.2.8.4 Notes

- ❑ If the given Gpio label is not related to the given Handle, the wm_ioSingleWrite function will fail.

3.7 BUS API

This API manages the I2C Soft and SPI bus operations.

3.7.1 Returned values definition

WM_BUS_PROC_DONE (0) : the function processing is successfuly done.
WM_BUS_MODE_UNKNOWN_TYPE (-1) : unknown open mode type.
WM_BUS_UNKNOWN_TYPE (-11) : unknown bus type.
WM_BUS_BAD_PARAMETER (-12) : a parameter has a not allowed value.
WM_BUS_SPI1_ALREADY_USED (-13) : the SPI bus is already opened.
WM_BUS_I2C_SOFT_ALREADY_USED (-15) : the I2C Soft bus is already opened.
WM_BUS_UNKNOWN_HANDLE (-21) : the handle used has an incorrect value.
WM_BUS_HANDLE_NOT_OPENED (-22) : no opened handle for this bus.
WM_BUS_NOT_CONNECTED_ON_I2C (-31) : no peripheral connected on I2C Soft bus.
WM_BUS_NOT_ALLOWED_ADDRESS (-32) : unknown address.
WM_BUS_I2C_SOFT_GPIO_NOT_GPIO (-33) : the function tried to Open I2C Soft bus with a GPI or a GPO.
WM_BUS_SPI_SIZE_TOO_LARGE (-36) : the function has tried to read or write more than 512 bytes on SPI bus.
WM_BUS_I2C_SIZE_TOO_LARGE (-37) : the function has tried to read or write more than 512 bytes on I2C bus.

3.7.2 The wm_busOpen Function

The wm_busOpen function allows to allocate a Handle on the required bus, and to open it for further read/write operations.

Its prototype is:

```
s32 wm_busOpen (                        u32 BusType,
                                        u32 Mode
                                        wm_busSettings_u * Settings );
```

3.7.2.1 Parameters

BusType:

Type of the bus to open. Defined values are:

- ❑ **WM_BUS_SPI1** for SPI bus ;
- ❑ **WM_BUS_SOFT_I2C** for I2C software bus.

Mode:

Bus mode ; the only defined value is WM_BUS_MODE_STANDARD.

Settings:

Pointer on settings union, defined as below.

```
typedef union
{
        wm_busSPISettings_t             SPI;
        wm_busI2CSoftSettings_t         I2C_Soft;
} wm_busSettings_u;
```

To open the SPI bus, you must use the SPI member of this union, defined as below:

```
typedef struct
{
        u32                          Clk_Speed;
        u32                          Clk_Mode;
} wm_busSPISettings_t;
```

The Clk_Speed parameter is the SPI clock speed ; defined values are:

❑ **WM_SCL_SPEED_101Khz ;**
❑ **WM_SCL_SPEED_812Khz ;**
❑ **WM_SCL_SPEED_1_625MHz ;**
❑ **WM_SCL_SPEED_3_25MHz.**

The Clk_Mode parameter is the SPI clock mode ; defined values are:

❑ **WM_SCK_MODE_0** (rest state 0, data valid on rising edge);
❑ **WM_SCK_MODE_1** (rest state 0, data valid on falling edge);
❑ **WM_SCK_MODE_2** (rest state 1, data valid on rising edge);
❑ **WM_SCK_MODE_3** (rest state 1, data valid on falling edge);

To open the I2C soft bus, you must use the I2C_Soft parameter of the union, defined as below:

```
typedef struct
{
        u32                          Scl_Gpio;
        u32                          Sda_Gpio;
} wm_busI2CSoftSettings_t;
```

The Scl_Gpio parameter is the label of the Gpio used to handle the SCL signal.
The Sda_Gpio parameter is the label of the Gpio used to handle the SDA signal.
Each of these labels must be a member of the wm_ioLabel_u union (see §3.6.2.1.2).

3.7.2.2 Returned Values

On successful completion, the function returns a positive or null Handle, to use for further Read / Write / Close operations on this bus.

Otherwise, the function will return a negative error value (cf §3.7.1 “Return values definition”).

3.7.2.3 Required Header

```
Wm_bus.h
```

3.7.2.4 Notes

❑ For I2C soft bus, the two Gpios labels passed in the Settings parameter must not be allocated by the embedded application ; only Gpio are allowed, using Gpi or Gpo to open the I2C bus will result as an error.

❑ A bus is available only if it was not opened before by AT commands (AT+WBM), otherwise, the wm_busOpen will result as an error. If a bus is opened by the Embedded application, it will be not available to AT commands, until the use of wm_busClose function.

3.7.3 The wm_busClose Function

The wm_busClose function allows to close a bus previously allocated by the wm_busOpen function.
Its prototype is:

```
s32 wm_busClose (                    s32 Handle );
```

3.7.3.1 Parameters

Handle:
Handle of the bus to close, returned by wm_busOpen function.

3.7.3.2 Returned Values

On successful completion, the function returns 0.

Otherwise, the function will return a negative error value (cf §3.7.1 "Return values definition").

3.7.3.3 Required Header

```
Wm_bus.h
```

3.7.3.4 Notes

- ❑ For I2C soft bus, the two Gpios labels passed in the Settings parameter of the wm_busOpen function are available again after the return of the wm_busClose function.

3.7.4 The wm_busWrite Function

The wm_busWrite function allows to write on a bus previously allocated by the wm_busOpen function.
Its prototype is:

```
s32 wm_busWrite (                    s32 Handle
                                     u32 Address,
                                     void * pDataToWrite,
                                     u32 NbBytes );
```

3.7.4.1 Parameters

Handle:
Handle of the bus device to write on, returned by wm_busOpen function.

Address:
Address of the device present on the requested bus, at which the function must write. This address depends on bus type:
For SPI: This parameter uses a set of chip select pins, dedicated to specific mapping of address:

- ❑ **WM_BUS_SPI_ADDRESS_NO_CS** : the function does not use any Chip Select (in order to use a GPIO as Chip Select, for example);
- ❑ **WM_BUS_SPI_ADDRESS_SPI_EN** : the function uses the SPI_EN pin as Chip Select ;
- ❑ **WM_BUS_SPI_ADDRESS_SPI_AUX** : the function uses the SPI_AUX pin as Chip Select.

For I2C soft: this parameter is the slave address byte. This is a 7-bits address, shift to left from 1 bit, padded with the LSB set to 0 (to write), and sent on the I2C bus before performing the writing operation.

pDataToWrite:
Buffer containing data to write on the requested bus.

NbBytes
Size of the pDataToWrite buffer. This size must not exceed 512 bytes.

3.7.4.2 Returned Values

On successful completion, the function returns the number of bytes written.

Otherwise, the function will return a negative error value (cf §3.7.1 “Return values definition”).

3.7.4.3 Required Header

```
Wm_bus.h
```

3.7.5 The wm_busRead Function

The wm_busRead function allows to read on a bus previously allocated by the wm_busOpen function.
Its prototype is :

```
s32 wm_busRead (                 s32 Handle
                                 u32 Address,
                                 void * pDataToRead,
                                 u32 NbBytes );
```

3.7.5.1 Parameters

Handle:
Handle of the bus device to read from, returned by wm_busOpen function.

Address:
Address of the device present on the requested bus, at which the function must read. This address depends on bus type:
For SPI: this parametrer uses a set of chip of select pins, dedicated to specific mapping of address:

- ❑ **WM_BUS_SPI_ADDRESS_NO_CS** : the function does not use any Chip Select (in order to use a GPIO as Chip Select, for example) ;
- ❑ **WM_BUS_SPI_ADDRESS_SPI_EN** : the function uses the SPI_EN pin as Chip Select ;
- ❑ **WM_BUS_SPI_ADDRESS_SPI_AUX** : the function uses the SPI_AUX pin as Chip Select.

For I2C soft: this parameter is the slave address byte. This is a 7-bits address, shift to left from 1 bit, padded with the LSB set to 1 (ro read), and sent on the I2C bus before performing the readintg operation.

pDataToRead:
Buffer containing data to read from the requested bus.

For SPI bus, the 2 first bytes should be used to send an operation code byte to the slave, before performing the reading operation. The first byte is the operation code length, in bits (from 1 to 8). The second byte is operation code value (as the MSB in always sent first, if the length is less than 8 bits, only the most significant bytes will be sent (example: to send first a bit set to 1, the buffer must be set to “0180“)).

NbBytes
Size of the pDataToRead buffer. This size must not exceed 512 bytes.

3.7.5.2 Returned Values

On successful completion, the function returns the number of bytes read.

Otherwise, the function will return a negative error value (cf §3.7.1 “Return values definition”).

3.7.5.3 Required Header

Wm_bus.h

3.8 Standard Library

The available standard functions are as follows:

char *	wm_strcpy	(char * *dst*, char * *src*);
char *	wm_strncpy	(char * *dst*, char * *src*, u32 *n*);
char *	wm_strcat	(char * *dst*, char * *src*);
char *	wm_strncat	(char * *dst*, char * *src*, u32 *n*);
u32	wm_strlen	(char * *str*);
s32	wm_strcmp	(char * *s1*, char * *s2*);
s32	wm_strncmp	(char * *s1*, char * *s2*, u32 *n*);
s32	wm_stricmp	(char * *s1*, char * *s2*);
s32	wm_strnicmp	(char * *s1*, char * *s2*, u32 *n*);
char *	wm_memset	(char * *dst*, char c, u32 *n*);
char *	wm_memcpy	(char * *dst*, char * src, u32 *n*);
s32	wm_memcmp	(char * *dst*, char * src, u32 *n*);
char *	wm_itoa	(s32 *a*, char * *szBuffer*);
s32	wm_atoi	(char * *p*);
s32	wm_strcmpi	(char * *dst*, char * *src*);
s32	wm_strnicmp	(char * first, char * last, u32 count);
char	wm_isascii	(char c);
char	wm_isdigit	(char c);

Required Header

wm_stdio.h

4 FUNCTIONING

There are three different functioning modes, depending on the type of application. They are described in the following paragraphs.

4.1 Standalone External Application

This mode corresponds to the standard operation mode: no Embedded Application is active.

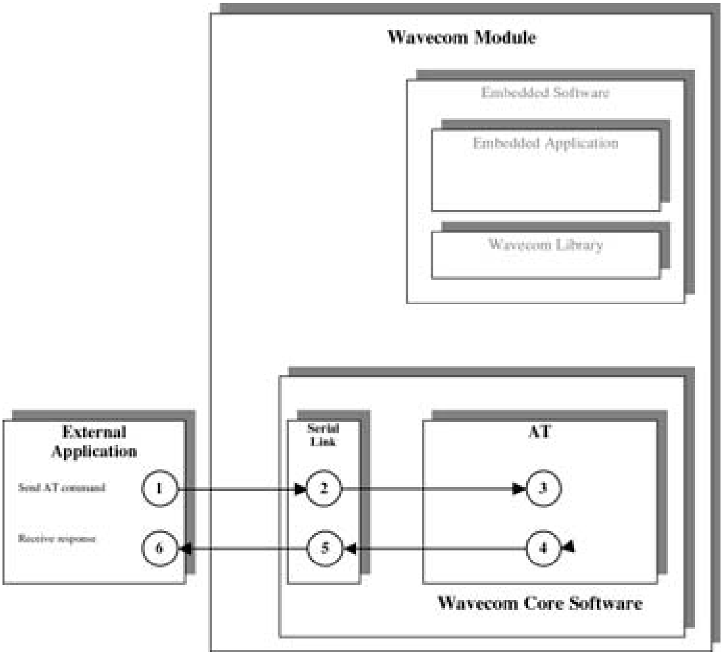


Figure 3: Standalone External Application Function

The steps are performed in the following sequence:

❶ The External Application sends an AT command,
❷ The serial link transmits the command to the AT processor function of the Wavecom Core Software,
❸ The AT function processes the command,
❹ The AT function sends an AT response to the External Application,
❺ This response is sent through the serial link, and
❻ The External Application receives the response.

Note: *This mode is also compatible with the mode described in § 4.2, where the AT function is in charge of dispatching the responses to the right application.*

4.2 Embedded Application in Standalone Mode

This mode is based on an Embedded Application driving the GSM product independently.

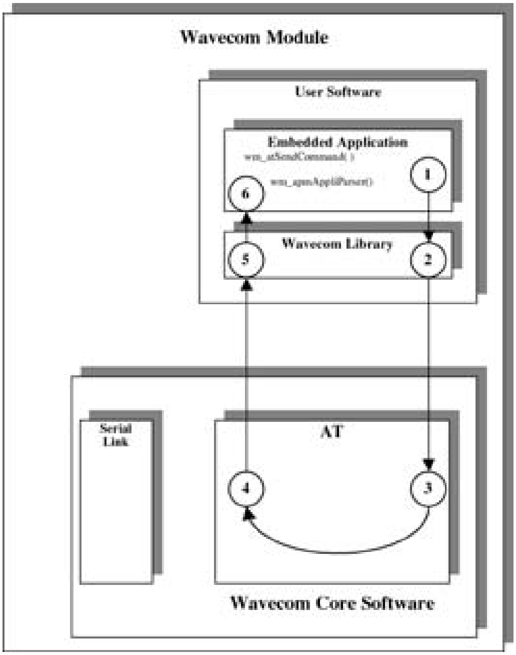


Figure 4: Embedded Application in Standalone Mode Function

The steps are performed in the following sequence:

❶ The Embedded Application calls the "wm_atSendCommand" function to send an AT command. The response parameter is then WM_AT_SEND_RSP_TO_EMBEDDED,

❷ The Wavecom library calls the appropriate AT function from the Wavecom Core Software,

❸ The AT function processes the command,

❹ The AT function sends the AT response to the Embedded Application,

❺ This response is dispatched by the Wavecom library which calls the "wm_apmAppliParser" function of the Embedded Application,

❻ The "wm_apmAppliParser" function processes the response (the AT response is a parameter of the function). The Message type is WM_AT_RESPONSE.

Example: appli.c file of a Standalone Mode embedded application

```
/***************************************************/
/*   Appli.c   -  Copyright Wavecom S.A. (c) 2001      */
/***************************************************/

#include "wm_types.h"
#include "wm_apm.h"

#define TIMER 01

/*************************/
/*  Mandatory Variables     */
/*************************/

char wm_apmCustomStack[1024];
const u16 wm_apmCustomStackSize = sizeof ( wm_apmCustomStack );

/*************************/
/*  Mandatory Functions     */
/*************************/

/**********************************/
/* wm_apmAppliInit                             */
/* Embedded Application initialisation */
/**********************************/
s32 wm_apmAppliInit ( wm_apmInitType_e InitType )
{
  wm_osDebugTrace(1, "Embedded: Appli Init" );

  wm_osStartTimer ( TIMER, FALSE, WM_S_TO_TICK ( 2 ) );
  return OK;
}
/*************************************/
/*  wm_apmAppliParser                           */
/* Embedded Application message parser   */
/*************************************/
s32 wm_apmAppliParser ( wm_apmMsg_t * pMessage )
{
  wm_osDebugTrace ( 1, "Embedded: Appli Parser" );

  switch ( pMessage->MsgTyp )
  {
    case WM_OS_TIMER:
      wm_osDebugTrace ( 1, "WM_OS_TIMER received" );
      if ( pMessage->Body.OSTimer.Ident == TIMER )
      {
        wm_atSendCommand ( 4, WM_AT_SEND_RSP_TO_EMBEDDED,
                                      “AT\r” );
        wm_osDebugTrace ( 1, "Send command \”AT\\r\”" );
      }
    break;

    case WM_AT_RESPONSE:
      wm_osDebugTrace ( 1, "WM_AT_RESPONSE received" );
      if ( pMessage->Body.ATResponse.Type ==
                            WM_AT_SEND_RSP_TO_EMBEDDED )
      {
        wm_osDebugTrace ( 1, "Response received:" );
        wm_osDebugTrace ( 1, pMessage->Body.ATResponse.StrData );
      }
    break;
  }

  return OK;
}
```

Target Monitoring Tool traces with this example:

```
Trace    CUS    1    Embedded: Appli Init
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_OS_TIMER received
Trace    CUS    1    Send command "AT\r"
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_AT_RESPONSE received
Trace    CUS    1    Response received:
Trace    CUS    1    <CR><LF>OK<CR><LF>
```

4.3 Cooperative Mode

This mode corresponds to the interaction between an External Application and an Embedded Application.

Whenever the Embedded Application wants to filter or spy **the commands** sent by the External Application, it can use the **command pre-parsing** mechanism.

Three types of subscription are available. They define the level of information required by the Embedded Application:

- The Embedded Application does not want to filter or spy the commands sent by the External Application: this is done using **WM_AT_CMD_PRE_WAVECOM_TREATMENT**.
- The Embedded Application wants to filter the AT commands sent by the External Application: this is done using **WM_AT_CMD_PRE_EMBEDDED_TREATMENT**. In this configuration, it is up to the Embedded Application to process or not the AT command and to send a response to the External Application.
- The Embedded Application wants only to spy the AT commands sent by the External Application: this is done using **WM_AT_CMD_PRE_BROADCAST.**

Whenever the Embedded Application wants to filter or spy the **responses** sent to the External Application, it can use the **response pre-parsing** mechanism.

Three types of subscription are available. They define the level of information required by the Embedded Application:

- The Embedded Application does not want to filter or spy the responses sent to the External Application: this is done using **WM_AT_RSP_PRE_WAVECOM_TREATMENT**.
- The Embedded Application wants to filter the AT responses sent to the External Application: this is done using **WM_AT_RSP_PRE_EMBEDDED_TREATMENT**. In this configuration, it is up to the Embedded Application to send a response to the External Application.
- The Embedded Application wants only to spy the AT responses sent to the External Application: this is done using **WM_AT_RSP_PRE_BROADCAST**.

4.3.1 Command Pre-Parsing Subscription Mechanism: WM_AT_CMD_PRE_EMBEDDED_TREATMENT

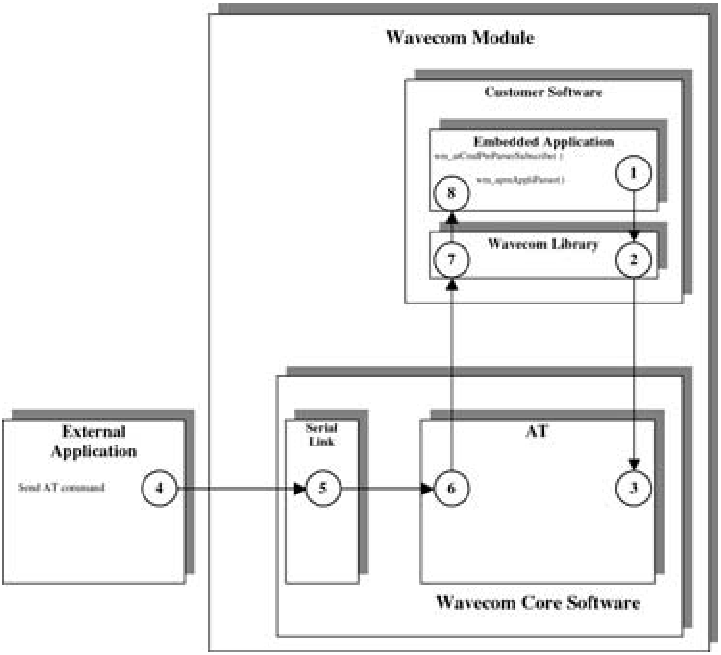


Figure 5: WM_AT_CMD_PRE_EMBEDDED_TREATMENT

The steps in a Pre-Parsing subscription are performed in the following sequence:

❶ The Embedded Application subscribes to the command pre-parsing service, by calling the wm_atCmdPreParserSubscribe() function,

❷ The Wavecom library calls the appropriate function from the Wavecom Core Software, and

❸ The AT function sets the subscription.

The steps in AT command processing are performed in the following sequence:

❹ The External Application sends an AT command,

❺ The serial link transmits the command to the AT processor function in the Wavecom Core Software,

❻ The AT function does not process the command but transmits it to the Embedded Application,

❼ The command is routed by the Wavecom library which calls the “wm_apmAppliParser” function of the Embedded Application (the Message type is WM_AT_CMD_PRE_PARSER),

❽ This function processes the command: the parameters of the function include the AT command and an indication that the command comes from an External Application.

Example: appli.c file of a WM_AT_CMD_PRE_EMBEDDED_TREATMENT Mode Embedded Application

```
/*************************************************/
/*    Appli.c   -  Copyright Wavecom S.A. (c) 2001      */
/*************************************************/
#include "wm_types.h"
#include "wm_apm.h"
#define TIMER 01
/***************************/
/*  Mandatory Variables      */
/***************************/
char wm_apmCustomStack[1024];
const u16 wm_apmCustomStackSize = sizeof ( wm_apmCustomStack );
/***************************/
/*  Mandatory Functions      */
/***************************/
/****************************************/
/*  wm_apmAppliInit                    */
/* Embedded Application initialisation */
/***************************************/
s32 wm_apmAppliInit ( wm_apmInitType_e InitType )
{
   wm_osDebugTrace(1, "Embedded: Appli Init" );
   wm_atCmdPreParserSubscribe ( WM_AT_CMD_PRE_EMBEDDED_TREATMENT );
   wm_osStartTimer ( TIMER, FALSE, WM_S_TO_TICK ( 2 ) );
   return OK;
}
/********************************************/
/*  wm_apmAppliParser                        */
/* Embedded Application message parser    */
/********************************************/
s32 wm_apmAppliParser ( wm_apmMsg_t * pMessage )
{
   wm_osDebugTrace ( 1, "Embedded: Appli Parser" );
   switch ( pMessage->MsgTyp )
   {
      case WM_OS_TIMER:
         wm_osDebugTrace ( 1, "WM_OS_TIMER received" );
      break;
      case WM_AT_CMD_PRE_PARSER:
         wm_osDebugTrace ( 1, "WM_AT_CMD_PRE_PARSER received" );
         if ( pMessage->Body.ATCmdPreParser.Type ==
                           WM_AT_CMD_PRE_EMBEDDED_TREATMENT )
         {
            wm_osDebugTrace ( 1, "command received:" );
            wm_osDebugTrace ( 1, pMessage->Body.ATCmdPreParser.StrData );

            if ( !wm_strncmp ( pMessage->Body.ATCmdPreParser.StrData,
                                    "AT-W", 4 ) )
            {
               /* filter Specific embedded application command */
               wm_osDebugTrace ( 1, "Specific embedded application command" );

               /* send response to external application */
               wm_atSendRspExternalApp ( 10, "\r\n->WOK\r\n" );
            }
            else
            {
               /* command must be treated by AT Software */
               wm_osDebugTrace ( 1, "Wavecom Core Software command" );
               wm_atSendCommand (
                   pMessage->Body.ATCmdPreParser.StrLength,
                  WM_AT_SEND_RSP_TO_EXTERNAL,
                  pMessage->Body.ATCmdPreParser.StrData );
            }
         }
      break;
   }

   return OK;
}
```

An AT command log for the external application with this example:

```
AT
OK
AT-W
->WOK
```

Target Monitoring Tool traces with this example:

```
Trace    CUS    1    Embedded: Appli Init
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_OS_TIMER received
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_AT_CMD_PRE_PARSER received
Trace    CUS    1    command received:
Trace    CUS    1    AT<CR>
Trace    CUS    1    Wavecom Core Software command
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_AT_CMD_PRE_PARSER received
Trace    CUS    1    command received:
Trace    CUS    1    AT-W<CR>
Trace    CUS    1    Specific embedded application command
```

4.3.2 Command Pre-Parsing Subscription Process: WM_AT_CMD_PRE_BROADCAST

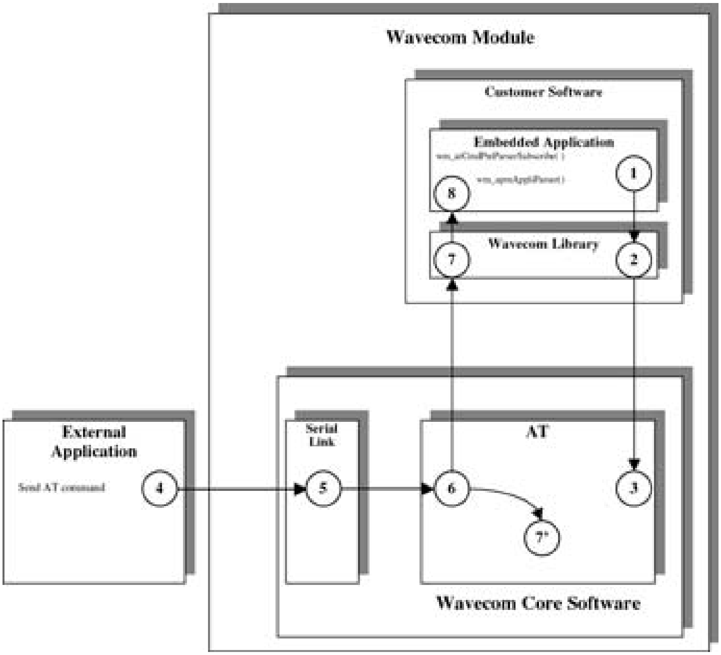


Figure 6: WM_AT_CMD_PRE_BROADCAST

The steps in a Pre-Parsing subscription are performed in the following sequence:

❶ The Embedded Application subscribes to the command pre-parsing service, by calling the wm_atCmdPreParserSubscribe() function,

❷ The Wavecom library calls the appropriate function in the Wavecom Core Software, and

❸ The AT function sets the subscription.

The steps in AT command processing are performed in the following sequence:

❹ The External Application sends an AT command,

❺ The serial link transmits the command to the AT function of the Wavecom Core Software,

❻ This AT function checks the subscription status of the “external” AT command,

❼ This external AT command is dispatched by the Wavecom library which calls the “wm_apmAppliParser” function of the Embedded Application,

❼’ Meanwhile, the AT function processes the command,

❽ The “wm_apmAppliParser” function spies the command: the parameters include the AT command and the indication of whether or not the command is a copy (the Message type is WM_AT_CMD_PRE_PARSER).

Example: appli.c file of a WM_AT_CMD_PRE_BROADCAST Mode embedded application

```
/**************************************************/
/*  Appli.c   -  Copyright Wavecom S.A. (c) 2001          */
/**************************************************/

#include "wm_types.h"
#include "wm_apm.h"

#define TIMER 01

/**************************/
/*  Mandatory Variables       */
/**************************/

char wm_apmCustomStack[1024];
const u16 wm_apmCustomStackSize = sizeof ( wm_apmCustomStack );

/**************************/
/*  Mandatory Functions       */
/**************************/

/*************************************/
/*  wm_apmAppliInit                               */
/* Embedded Application initialisation */
/*************************************/
s32 wm_apmAppliInit ( wm_apmInitType_e InitType )
{
   wm_osDebugTrace(1, "Embedded: Appli Init" );
   wm_atCmdPreParserSubscribe ( WM_AT_CMD_PRE_BROADCAST );
   wm_osStartTimer ( TIMER, FALSE, WM_S_TO_TICK ( 2 ) );
   return OK;
}
/*****************************************/
/*  wm_apmAppliParser                                 */
/* Embedded Application message parser   */
/*****************************************/
s32 wm_apmAppliParser ( wm_apmMsg_t * pMessage )
{
   wm_osDebugTrace ( 1, "Embedded: Appli Parser" );

   switch ( pMessage->MsgTyp )
   {
      case WM_OS_TIMER:
         wm_osDebugTrace ( 1, "WM_OS_TIMER received" );
      break;

      case WM_AT_CMD_PRE_PARSER:
         wm_osDebugTrace ( 1, "WM_AT_CMD_PRE_PARSER received" );
         if ( pMessage->Body.ATCmdPreParser.Type ==
                                    WM_AT_CMD_PRE_BROADCAST )
         {
            /* spy command sent by external application */
           wm_osDebugTrace ( 1, "command received from external application" );
            wm_osDebugTrace ( 1, pMessage->Body.ATCmdPreParser.StrData );
         }
      break;
   }

   return OK;
}
```

AT command log for the external application with this example:

```
AT
OK
```

Target Monitoring Tool traces with this example:

```
Trace    CUS    1    Embedded: Appli Init
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_OS_TIMER received
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_AT_CMD_PRE_PARSER received
Trace    CUS    1    command received from external application
Trace    CUS    1    at<CR>
```

4.3.3 Response Pre-Parsing Subscription Process: WM_AT_RSP_PRE_EMBEDDED_TREATMENT

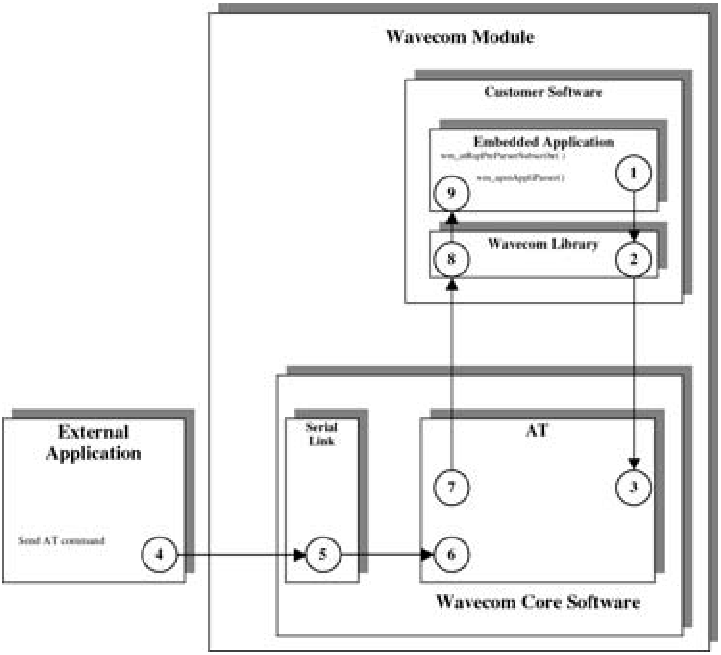


Figure 7: WM_AT_RSP_PRE_EMBEDDED_TREATMENT

The steps in a Pre-Parsing subscription are performed in the following sequence:

❶ The Embedded Application subscribes to the response pre-parsing facility, by calling the wm_atRspPreParserSubscribe() function,

❷ The Wavecom library calls the appropriate function from the Wavecom Core Software, and

❸ The AT function sets the subscription.

The steps in AT command processing are performed in the following sequence:

❹ The External Application sends an AT command,

❺ The serial link transmits the command to the AT function of the Wavecom Core Software,

❻ This configuration does not rely on command pre-parsing. The AT function processes the command,

❼ The AT function checks the subscription status of the response and does not send the response to the External Application. Instead, it sends the response to the Embedded Application,

❽ The response is dispatched by the Wavecom library which calls the "wm_apmAppliParser" function of the Embedded Application (the Message type is WM_AT_RSP_PRE_PARSER),

❾ This function processes the response (the parameters of the function include an indication of the response filtering).

Example: appli.c file of a WM_AT_RSP_PRE_EMBEDDED_TREATMENT Mode embedded application

```c
/*************************************************/
/*  Appli.c   -  Copyright Wavecom S.A. (c) 2001        */
/*************************************************/

#include "wm_types.h"
#include "wm_apm.h"
#define TIMER 01

/***************************/
/*  Mandatory Variables      */
/***************************/

char wm_apmCustomStack[1024];
const u16 wm_apmCustomStackSize = sizeof ( wm_apmCustomStack );

/***************************/
/*  Mandatory Functions      */
/***************************/

/*************************************/
/*  wm_apmAppliInit                             */
/* Embedded Application initialisation  */
/*************************************/
s32 wm_apmAppliInit ( wm_apmInitType_e InitType )
{
   wm_osDebugTrace(1, "Embedded: Appli Init" );
   wm_atRspPreParserSubscribe ( WM_AT_RSP_PRE_EMBEDDED_TREATMENT );
   wm_osStartTimer ( TIMER, FALSE, WM_S_TO_TICK ( 2 ) );
   return OK;
}
/****************************************/
/*  wm_apmAppliParser                              */
/* Embedded Application message parser   */
/****************************************/
s32 wm_apmAppliParser ( wm_apmMsg_t * pMessage )
{
   wm_osDebugTrace ( 1, "Embedded: Appli Parser" );

   switch ( pMessage->MsgTyp )
   {
      case WM_OS_TIMER:
         wm_osDebugTrace ( 1, "WM_OS_TIMER received" );
      break;

      case WM_AT_RSP_PRE_PARSER:
         wm_osDebugTrace ( 1, "WM_AT_RSP_PRE_PARSER received" );
         wm_osDebugTrace ( 1, pMessage->Body.ATRspPreParser.StrData );

         if ( pMessage->Body.ATRspPreParser.Type ==
                           WM_AT_RSP_PRE_EMBEDDED_TREATMENT )
         {
            if ( !wm_strncmp ( "\r\nOK\r\n",
                                 pMessage->Body.ATRspPreParser.StrData, 6 ) )
            {
               wm_osDebugTrace ( 1, "OK response modified for external application" );
               wm_atSendRspExternalApp ( 10, "\r\n->WOK\r\n" );
            }
            else
            {
               wm_osDebugTrace ( 1, "no modified response" );
               wm_atSendRspExternalApp (
                                 pMessage->Body.ATRspPreParser.StrLength,
                                 pMessage->Body.ATRspPreParser.StrData );
            }
         }
      break;
   }

   return OK;
}
```

AT commands log for the external application with this example:

```
AT
->WOK
at+wopen?
+WOPEN: 1
->WOK
```

Target Monitoring Tool traces with this example:

```
Trace    CUS    1    Embedded: Appli Init
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_OS_TIMER received
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_AT_RSP_PRE_PARSER received
Trace    CUS    1    <CR><LF>OK<CR><LF>
Trace    CUS    1    OK response modified for external application
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_AT_RSP_PRE_PARSER received
Trace    CUS    1    <CR><LF>+WOPEN: 1<CR><LF>
Trace    CUS    1    no modified response
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_AT_RSP_PRE_PARSER received
Trace    CUS    1    <CR><LF>OK<CR><LF>
Trace    CUS    1    OK response modified for external application
```

4.3.4 Response Pre-Parsing Subscription Process: WM_AT_RSP_PRE_BROADCAST

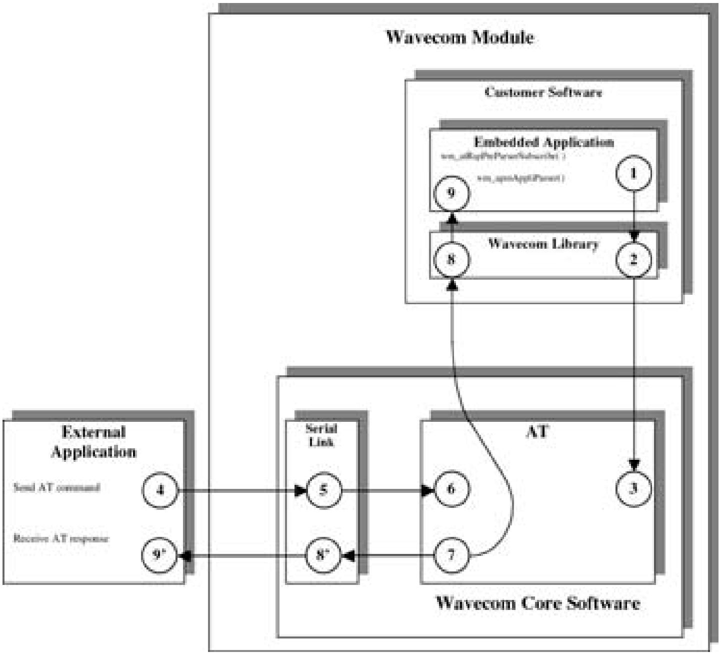


Figure 8: WM_AT_RSP_PRE_BROADCAST

The steps in a Pre-Parsing subscription are performed in the following sequence:

❶ The Embedded Application subscribes to the response pre-parsing facility, by calling the wm_atRspPreParserSubscribe() function,

❷ The Wavecom library calls the appropriate function in the Wavecom Core Software, and

❸ The AT function sets the subscription.

The steps in AT command processing are performed in the following sequence:

❹ The External Application sends an AT command,

❺ The serial link transmits the command to the AT function of the Wavecom Core Software,

❻ This configuration does not rely on command pre-parsing. The AT function processes the command,

❼ The AT function checks the subscription status of the response and sends it to both the External Application and the Embedded Application,

❽ The response is dispatched by the Wavecom library, which calls the "wm_apmAppliParser" function of the Embedded Application (the Message type is WM_AT_RSP_PRE_PARSER),

❾ This function processes the response (the parameters of the function include a broadcast response indication),

❽' This response is sent through the serial link,

❾' The External Application receives the response.

Example: appli.c file of a WM_AT_RSP_PRE_BROADCAST Mode embedded application

```
/*************************************************/
/*  Appli.c   -  Copyright Wavecom S.A. (c) 2001         */
/*************************************************/

#include "wm_types.h"
#include "wm_apm.h"
#define TIMER 01

/****************************/
/*  Mandatory Variables      */
/****************************/

char wm_apmCustomStack[1024];
const u16 wm_apmCustomStackSize = sizeof ( wm_apmCustomStack );

/****************************/
/*  Mandatory Functions      */
/****************************/

/****************************************/
/*  wm_apmAppliInit                              */
/* Embedded Application initialisation */
/****************************************/
s32 wm_apmAppliInit ( wm_apmInitType_e InitType )
{
   wm_osDebugTrace(1, "Embedded: Appli Init" );
   wm_atRspPreParserSubscribe ( WM_AT_RSP_PRE_BROADCAST );
   wm_osStartTimer ( TIMER, FALSE, WM_S_TO_TICK ( 2 ) );
   return OK;
}
/********************************************/
/*  wm_apmAppliParser                               */
/* Embedded Application message parser   */
/********************************************/
s32 wm_apmAppliParser ( wm_apmMsg_t * pMessage )
{
   wm_osDebugTrace ( 1, "Embedded: Appli Parser" );

   switch ( pMessage->MsgTyp )
   {
      case WM_OS_TIMER:
         wm_osDebugTrace ( 1, "WM_OS_TIMER received" );
      break;

      case WM_AT_RSP_PRE_PARSER:
         wm_osDebugTrace ( 1, "WM_AT_RSP_PRE_PARSER received" );

         if ( pMessage->Body.ATRspPreParser.Type ==
                                    WM_AT_RSP_PRE_BROADCAST )
         {
            /* spy response sent to external application */
            wm_osDebugTrace ( 1, "response sent to external application" );
            wm_osDebugTrace ( 1, pMessage->Body.ATRspPreParser.StrData );
         }
      break;
   }

   return OK;
}
```

AT command log for the external application with this example:

```
AT
OK
```

Target Monitoring Tool traces with this example:

```
Trace    CUS    1    Embedded: Appli Init
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_OS_TIMER received
Trace    CUS    1    Embedded: Appli Parser
Trace    CUS    1    WM_AT_RSP_PRE_PARSER received
Trace    CUS    1    response sent to external application
Trace    CUS    1    <CR><LF>OK<CR><LF>
```

4.3.5 Example: Embedded Application Using the Different Functioning Modes

```
/*************************************************/
/*  Appli.c   -  Copyright Wavecom S.A. (c) 2001          */
/*************************************************/

#include "wm_types.h"
#include "wm_apm.h"

#define TIMER 01

typedef enum
{
   STANDALONE,
   CMD_PREPARSING_EMBEDDED,
   CMD_PREPARSING_BROADCAST,
   RSP_PREPARSING_EMBEDDED,
   RSP_PREPARSING_BROADCAST,
} wm_AtMode_e;

/**************************/
/*  Mandatory Variables      */
/**************************/

char wm_apmCustomStack[1024];
const u16 wm_apmCustomStackSize = sizeof ( wm_apmCustomStack );

/**************************/
/*  Global Variables             */
/**************************/

wm_AtMode_e AtMode = STANDALONE;

/**************************/
/*  Global Function              */
/**************************/

void AtAutomate(state)
{
 switch(state)
 {
 case STANDALONE:
  wm_osDebugTrace(1, "STANDALONE" );
  wm_atCmdPreParserSubscribe(WM_AT_CMD_PRE_WAVECOM_TREATMENT);
  wm_atRspPreParserSubscribe(WM_AT_RSP_PRE_WAVECOM_TREATMENT);
  wm_atSendRspExternalApp(16,"STANDALONE mode");
  wm_atSendRspExternalApp(18,"send an at command");
 break;

 case CMD_PREPARSING_EMBEDDED:
  wm_osDebugTrace(1, "CMD_PREPARSING_EMBEDDED" );
  wm_atCmdPreParserSubscribe(WM_AT_CMD_PRE_EMBEDDED_TREATMENT);
  wm_atRspPreParserSubscribe(WM_AT_RSP_PRE_WAVECOM_TREATMENT);
  wm_atSendRspExternalApp(29,"CMD_PREPARSING_EMBEDDED mode");
  wm_atSendRspExternalApp(18,"send an at command");
 break;

 case CMD_PREPARSING_BROADCAST:
  wm_osDebugTrace(1, "CMD_PREPARSING_BROADCAST" );
  wm_atCmdPreParserSubscribe(WM_AT_CMD_PRE_BROADCAST);
  wm_atRspPreParserSubscribe(WM_AT_RSP_PRE_WAVECOM_TREATMENT);
  wm_atSendRspExternalApp(30,"CMD_PREPARSING_BROADCAST mode");
  wm_atSendRspExternalApp(18,"send an at command");
 break;

 case RSP_PREPARSING_EMBEDDED:
  wm_osDebugTrace(1, "RSP_PREPARSING_EMBEDDED" );
  wm_atCmdPreParserSubscribe(WM_AT_CMD_PRE_WAVECOM_TREATMENT);
  wm_atRspPreParserSubscribe(WM_AT_RSP_PRE_EMBEDDED_TREATMENT);
  wm_atSendRspExternalApp(29,"RSP_PREPARSING_EMBEDDED mode");
  wm_atSendRspExternalApp(18,"send an at command");
 break;

 case RSP_PREPARSING_BROADCAST:
  wm_osDebugTrace(1, "RSP_PREPARSING_BROADCAST" );
  wm_atCmdPreParserSubscribe(WM_AT_CMD_PRE_WAVECOM_TREATMENT);
  wm_atRspPreParserSubscribe(WM_AT_RSP_PRE_BROADCAST );
  wm_atSendRspExternalApp(30,"RSP_PREPARSING_BROADCAST mode");
  wm_atSendRspExternalApp(18,"send an at command");
 break;
```

```
  default:
   wm_osDebugTrace(1, "mode unexpected" );
   break;
  }
}
/**************************/
/*  Mandatory Functions      */
/**************************/
/*************************************/
/*  wm_apmAppliInit                          */
/* Embedded Application initialisation  */
/*************************************/
s32 wm_apmAppliInit ( wm_apmInitType_e InitType )
{
   wm_osDebugTrace(1, "Embedded: Appli Init" );
   wm_osStartTimer ( TIMER, FALSE, WM_S_TO_TICK ( 2 ) );
   return OK;
}
 /****************************************/
/*  wm_apmAppliParser                          */
/* Embedded Application message parser   */
/****************************************/
bool wm_apmAppliParser ( wm_apmMsg_t * pMessage )
{
   wm_osDebugTrace ( 1, "Embedded: Appli Parser" );
   switch ( pMessage->MsgTyp )
   {
      case WM_OS_TIMER:
         wm_osDebugTrace ( 1, "WM_OS_TIMER received" );
         AtAutomate(AtMode);
         if (AtMode!=RSP_PREPARSING_BROADCAST)
         {
            AtMode++;
            wm_osStartTimer (TIMER, FALSE, WM_S_TO_TICK(10));
         }
      break;
      case WM_AT_RESPONSE:
         wm_atSendRspExternalApp( 33, "message WM_AT_RESPONSE
                                        received:");
         wm_strncpy(strReceived, pMessage->Body.ATResponse.StrData,
                              pMessage->Body.ATResponse.StrLength);
         strReceived[pMessage->Body.ATResponse.StrLength] = '\0';
         wm_atSendRspExternalApp( pMessage->Body.ATResponse.StrLength +
                                      1, strReceived );
      break;
      case WM_AT_CMD_PRE_PARSER:
         wm_atSendRspExternalApp(39, "message WM_AT_CMD_PRE_PARSER
                                        received:");
         wm_strncpy(strReceived, pMessage->Body.ATCmdPreParser.StrData,
                              pMessage->Body.ATCmdPreParser.StrLength);
         strReceived[pMessage->Body.ATCmdPreParser.StrLength] = '\0';
         wm_atSendRspExternalApp(pMessage->Body.ATResponse.StrLength +
                                      1, strReceived );
      break;

      case WM_AT_RSP_PRE_PARSER:
         wm_atSendRspExternalApp(39, "message WM_AT_RSP_PRE_PARSER
                                              received:");
         wm_strncpy(strReceived, pMessage->Body.ATRspPreParser.StrData,
                                  pMessage->Body.ATRspPreParser.StrLength);
         strReceived[pMessage->Body.ATRspPreParser.StrLength] = '\0';
         wm_atSendRspExternalApp(pMessage->Body.ATResponse.StrLength +
                                          1, strReceived );
      break;
   }
   return TRUE;
}
```

AT command log for the external application with this example:

```
STANDALONE mode
             at                                         no interaction between external
             OK                                              and embedded application

CMD_PREPARSING_EMBEDDED mode
send an at command
             at                                    command sent to embedded application
• message WM_AT_CMD_PRE_PARSER received:
             at                                       and not to Wavecom AT Software

CMD_PREPARSING_BROADCAST mode
send an at command
             at                                                   command sent to both
             OK                                       response of Wavecom AT Software
• message WM_AT_CMD_PRE_PARSER received:
             at                               command received by embedded application

RSP_PREPARSING_EMBEDDED mode
send an at command
             at                                    command sent to Wavecom AT Software
• message WM_AT_RSP_PRE_PARSER received:
             OK                                   response sent to embedded application

RSP_PREPARSING_BROADCAST mode
send an at command
             at                                    command sent to Wavecom AT Software
             OK                                   response sent to external application
• message WM_AT_RSP_PRE_PARSER received:
             OK                                   response sent to embedded application
```

Target Monitoring Tool traces with this example:

```
Trace    CUS    1        Embedded: Appli Init
Trace    CUS    1        Embedded: Appli Parser
Trace    CUS    1        WM_OS_TIMER received
Trace    CUS    1        STANDALONE
Trace    CUS    1        Embedded: Appli Parser
Trace    CUS    1        WM_OS_TIMER received
Trace    CUS    1        CMD_PREPARSING_EMBEDDED
Trace    CUS    1        Embedded: Appli Parser
Trace    CUS    1        Embedded: Appli Parser
Trace    CUS    1        WM_OS_TIMER received
Trace    CUS    1        CMD_PREPARSING_BROADCAST
Trace    CUS    1        Embedded: Appli Parser
Trace    CUS    1        Embedded: Appli Parser
Trace    CUS    1        WM_OS_TIMER received
Trace    CUS    1        RSP_PREPARSING_EMBEDDED
Trace    CUS    1        Embedded: Appli Parser
Trace    CUS    1        Embedded: Appli Parser
Trace    CUS    1        WM_OS_TIMER received
Trace    CUS    1        RSP_PREPARSING_BROADCAST
Trace    CUS    1        Embedded: Appli Parser
```

WAVECOM S.A. - 12, boulevard Garibaldi - 92442 Issy-les-Moulineaux Cedex - France - Tel: +33 (0)1 46 29 08 00 - Fax: +33 (0)1 46 29 08 08
WAVECOM, Inc. - 4810 Eastgate Mall - Second floor - San Diego, CA 92121 - USA - Tel: +1 858 362 0101 - Fax: +1 858 558 5485
WAVECOM Asia Pacific Ltd - 5/F Shui On Centre - 6/8 Harbour Road - Hong Kong, PRC - Tel: +852 2824 0254 - Fax: +852 2824 0255

www.wavecom.com