Connectivity Development Guide

Internet Library 5.54

SIERRA WIRELESS

4111845
2.0
May 30, 2012

Important Notice

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

Safety and Hazards

Do not operate the Sierra Wireless modem in areas where cellular modems are not advised without proper device certifications. These areas include environments where cellular radio can interfere such as explosive atmospheres, medical equipment, or any other equipment which may be susceptible to any form of radio interference. The Sierra Wireless modem can transmit signals that could interfere with this equipment. Do not operate the Sierra Wireless modem in any aircraft, whether the aircraft is on the ground or in flight. In aircraft, the Sierra Wireless modem **MUST BE POWERED OFF**. When operating, the Sierra Wireless modem can transmit signals that could interfere with various onboard systems.

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

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

Limitations of Liability

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

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

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

Customer understands that Sierra Wireless is not providing cellular or GPS (including A-GPS) services. These services are provided by a third party and should be purchased directly by the Customer.

SPECIFIC DISCLAIMERS OF LIABILITY: CUSTOMER RECOGNIZES AND ACKNOWLEDGES SIERRA WIRELESS IS NOT RESPONSIBLE FOR AND SHALL NOT BE HELD LIABLE FOR ANY DEFECT OR DEFICIENCY OF ANY KIND OF CELLULAR OR GPS (INCLUDING A-GPS) SERVICES.

Patents

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

This product includes technology licensed from QUALCOMM®.

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

Copyright

Trademarks

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

, ®, inSIM®, WAVECOM®, WISMO®, Wireless Microprocessor®, Wireless CPU®, Open AT® are filed or registered trademarks of Sierra Wireless S.A. in France and/or in other countries.

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

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

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

Other trademarks are the property of the respective owners.

Contact Information

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

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

Document History

Version	Date	Updates
001	January 11, 2012	Creation based on Internet Library 5.43 for Open AT Application Framework 2.50.
2.0	May 30, 2012	Updated document legal boilerplate. Added the following clarifications about multitasking feature: • A note in the wip_netInit section, • A note in the IP Bearer Management section, and • An update to the Multitasking Feature section.

Contents

1. Introduction

The aim of this document is to provide Sierra Wireless customers with a full description of the APIs associated with the IP Connectivity library.

1.1. Abbreviations and Glossary

Abbreviation	Definition
ADL	Application Development Layer
API	Application Programming Interface
APN	Access Point Name
ASN	Abstract Syntax Notation
AT	Attention
BSD	Berkeley Software Distribution
CHAP	Challenge Handshake Authentication Protocol
CID	Context Identifier
DNS	Domain Name Service
EDGE	Enhanced Data rates for GSM Evolution
FTP	File Transfer Protocol
GGSN	Gateway GPRS Support Node
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communication
HTTP	Hyper Text Transfer Protocol
ICMP	Internet Control Message Protocol
IGMP	Internet Group Management Protocol
IMAP	Internet Message Access Protocol
IN/OUT/GLB	In, Out or Global. See Glossary.
IP	Internet Protocol
IPCP	Internet Protocol Control Protocol
IPv4	Internet Protocol Version 4
LAN	Local Area Network
LCP	Link Control Protocol
M	Mandatory
MMS	Multimedia Message Service
MS-CHAP	Microsoft Challenge Handshake Authentication
MS	Mobile Station
MSS	Maximum Segment Size
NA	Not Applicable
NU	Not Used
O	Optional
OID	Object Identifier
PAP	Password Authentication Protocol
PDP	Packet Data Protocol

Abbreviation	Definition
POP3	Post Office Protocol
POSIX	Portable Operating System Interface
PPP	Point-to-Point Protocol
RFC	Request For Comments
SMS	Short Messaging Service
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
TOS	Type Of Service
TTL	Time To Live
UART	Universal Asynchronous Receiver Transmitter
UDP	User Data Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USB	Universal Serial Bus
WIFI	Wireless Fidelity
3G	The third generation of developments in wireless technology

1.2. Glossary

In/out/Glb: used in function parameters:

- “In” if the parameter is given to the function
- “Out” if the parameter is the result of the function
- “Glb” (for Global) if the parameter is used for both

2. Global Architecture

2.1. Concepts

A network operation involves reading and writing data through channels. Once a channel is properly opened and set up, reading and writing through it is largely protocol independent.

Sierra Wireless provides a generic, high-level API that abstracts the underlying protocols of communication channels. This API relies on the following key concepts:

Channels are opaque data which represent a means of communication; for example, an open and connected socket. This interface could be reused for other protocols such as X -MODEM over an UART, SMS over GSM.

Events, being single-threaded, need non-blocking operations. The channels have a callback function registered with them, which describe how to react to noteworthy events, mainly read, write, close and an error.

Options are used to provide user defined configurations. The APIs are available in two formats.

APIs with no options (BASIC): These APIs uses default settings. For example, wip_netInit API is used to initialize the Internet Library with default settings.

APIs with options (OPT): These APIs accept a series of variable arguments of the form (OPTION_ID_0, optionValue_0, OPTION_ID_n, optionValue_n, END_MARKER) and are used to configure with user defined settings .Note that the options provided by the user will be checked at runtime for consistency.

The channels that are implemented to support IP are:

- TCP server sockets
- TCP communication sockets
- UDP sockets (communication sockets, as there is no notion of server in UDP)
- ICMP/Ping sockets

2.2. Feature Description

Sierra Wireless customers are provided with an advanced set of APIs that give them complete IP connectivity control. This allows an application to communicate using IP connectivity on different types of bearers (UART, GSM, GPRS, and EDGE) simultaneously.

Figure 1. Communication between Four Equipments

Notice that embedded module #1(the one on the left) has two IP addresses, one for each link.

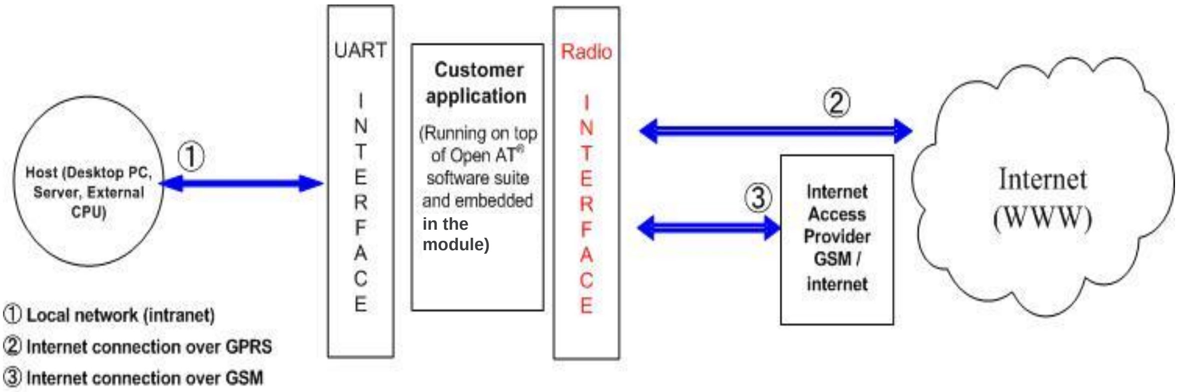


Figure 2. Uses of the New IP Stack (Use Cases 2 and 3 are Exclusive)

The Sierra Wireless Software Suite also supports ‘pure’ IP APIs which can provide better capabilities and control.

The socket abstraction layer gives high-level access to communication abilities, through a channel and its dedicated API. The following types of channels are implemented:

- a TCP channel implementation, which allows users to create and use client and server TCP sockets
- a UDP channel implementation, which allows users to create and use UDP sockets
- a PING channel implementation, which allows users to configure and send ICMP ECHO requests, or “pings”, and to receive feedback on response times, routing errors or timeout errors

The bearers are handled by the bearer manager which provides IP connectivity using various links. Several bearers can be activated simultaneously. The following links are currently supported:

- GSM data
- GPRS
- direct connection on an UART
- Ethernet bearer

Features of the TCP/IP protocol Stack include:

- IP, ICMP, UDP, TCP Protocols

- all RFC 1122 requirements for host-to-host interoperability
- fragmentation and reassembly of IP datagrams
- support for multiple network interfaces (forwarding of packets between interfaces is not enabled by default)
- loopback interface

Socket Layer:

- configuration of socket receive and send buffers
- control of some IP header fields such as TTL, TOS, "Don’t fragment" flag

TCP Sockets:

- congestion control (slow start, congestion avoidance, fast retransmit and fast recovery)
- option for disabling the Naggle algorithm
- immediate notification of all connection state changes
- support for normal connection termination and reset of the connection

DNS Resolver:

- integrated into the socket abstraction layer
- support for primary and secondary DNS servers

The PPP is required by GSM and UART bearers, the following features are supported:

- client and server mode
- authentication using PAP, CHAP, MS-CHAPv1 or MS-CHAPv2
- auto-configuration of IP address, primary and secondary DNS servers

2.3. New Interface

The new version of the IP stack provides a rich and simple user interface. The advantages of this new interface are as follows:

- clearly distinguishes the management of the bearer (GSM/GPRS) from the IP sockets management
- provides the user with the flexibility to configure and set IP related parameters. For example, during configuration of the bearer using PPP protocol, the user can select different authentication mechanisms such as PAP, CHAP/MS_CHAP
- provides an interface to configure the maximum number of sockets that can be used by the customer application
- allows the customer application to manage the socket dynamically (BSD-like interface)

2.4. Use Cases

This feature can be used by all users who communicate with IP, using GPRS, serial links, or any IP-compatible physical peripherals (WIFI, Ethernet) or radio bearers (EDGE, 3G) supported by Sierra Wireless intelligent embedded module.

The channel abstraction can also be used to encapsulate all kinds of network-oriented protocols such as X-MODEM, FTP, HTTP, POP, IMAP and SMS. With the uniform channel API, an application can change the communication channel it uses easily without any modification of its source code (except channel opening).

2.5. Channels Logical Hierarchy

Although there is no native support for object-oriented inheritance in C, different channels implementing various services are related to one another in terms of the services they support. These channels support a minimal number of common APIs which include creation, closing, reaction to events, and advanced configuration option lists. Most of the channels additionally support read and write operations. Many future channel types support concurrent download and upload of data, identified by a resource string: FTP, HTTP, IMAP, POP and access to local file system. These APIs defined as successive extensions should be seen as refinements of channel types and subtypes. To present them, we will specify abstract channel types, which introduce these APIs; actual protocols will be concrete implementations of these abstract interfaces.

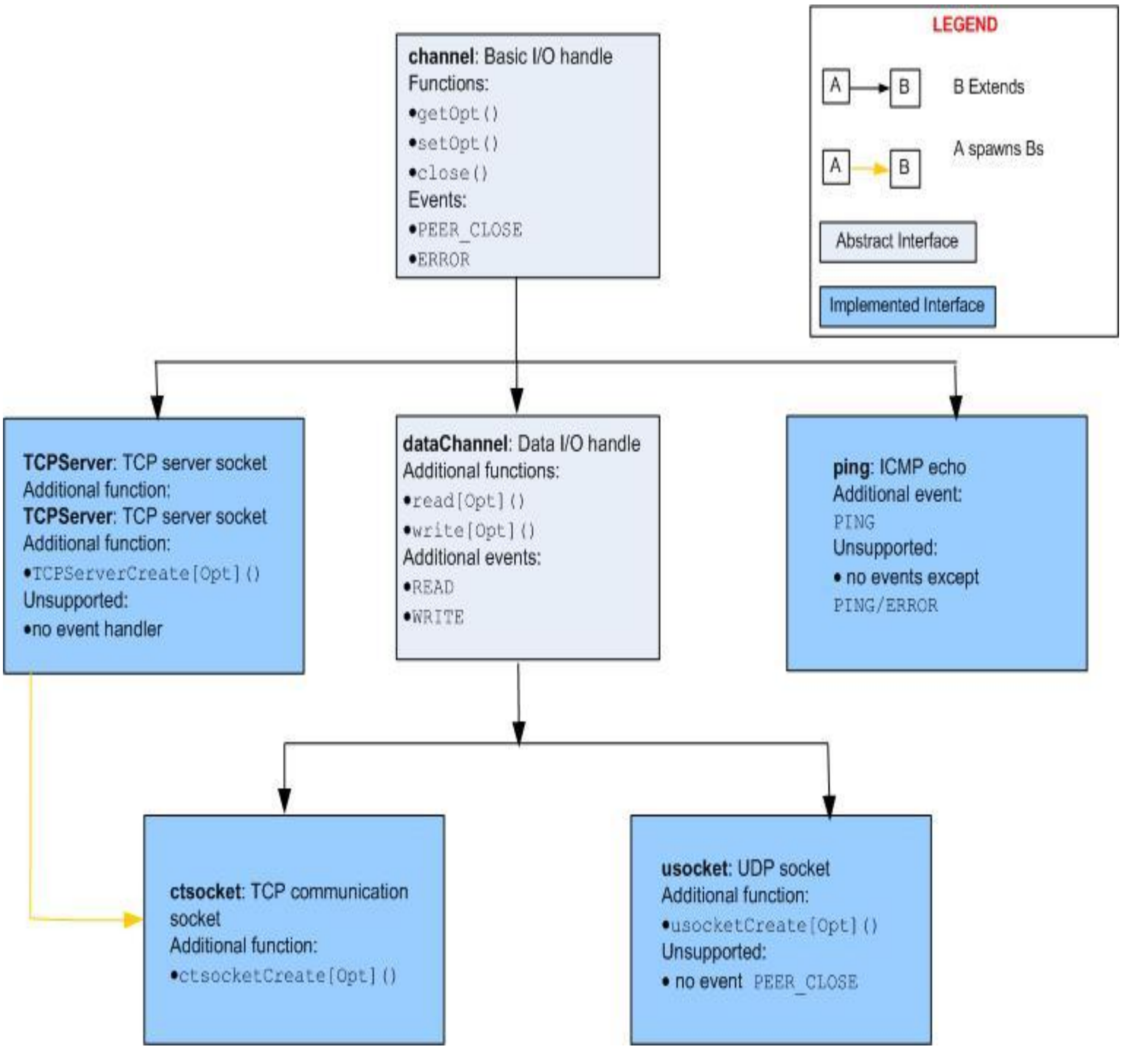


Figure 3. Channel Classes Hierarchy

2.5.1. Channel: Abstract, Basic I/O Handle

This channel supports the getOpts, setOpts and close operations. There is no real implementation of a channel; it is only the common interface for actual protocols.

Events that are supported by this channel include WIP_CEV_PEER_CLOSE and ERROR. ERROR has an errno number and an error message as parameters.

2.5.2. Data Channel: Abstract Data Transfer Handle

This is also an abstract channel type. It supports functions such as read, readOpts, write, writeOpts, as well as channel functions (close, getOpts, setOpts).

It supports events such as:

- READ (data has arrived)
- WRITE (buffer space has been freed to send some data)
- channel events

READ has an u32 readable field indicating the number of readable bytes, and WRITE has an u32 writable field which indicates how much data can be written. As a specialization of channel, it also supports the event WIP_CEV_PEER_CLOSE.

2.5.3. TCPServer: Server TCP Socket

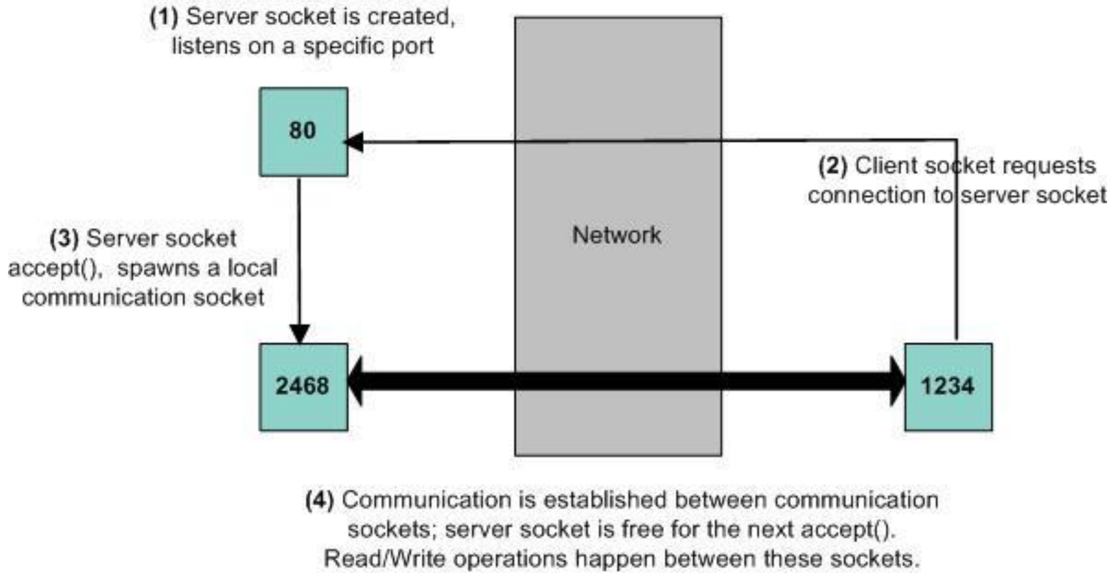


Figure 4. TCP Socket Spawning Process

TCPServer does not have a specialized dataChannel; it neither supports read nor supports write. Its purpose is to listen for connection requests, accept them, and spawn a TCP communication socket peered with the one that requested the communication. TCPServers supports create, getopt, setopt and close operations.

Spawning

Spawning a communication is a common POSIX pattern. A globally known server channel creates secondary, communication channels. In the TCP server case, a server TCP socket listens on a familiar port such as 80 for HTTP and 21 for FTP. Whenever a remote socket contacts the server socket, a communication is established between the client socket and a specially created socket on the server side, which is spawned by the server socket. A direct communication between the server and the client socket must be avoided, as that would monopolize the server socket.

2.5.4. TCPClient: Communication TCP Socket

TCPClients read and write a reliable and ordered byte stream. In addition to the dataChannel interface it inherits from, it supports creation through wip_TCPClientCreate[Opts]() (creation can also happen through Spawning by TCPServer, equivalent of BSD's accept()) it also supports the Abort() and Shutdown() functions.

Creation of TCP clients can happen due to local creation and connection requests on a remote server socket. This includes:

- creating the socket
- connecting it to a host through a server socket
- setting up a callback to react to network events happening to the socket

All of this happens at once in a single wip_TCPServerCreate() API call, so that the user is not exposed to partially configured communication sockets that are not yet in a usable state. As soon as it is created, the socket is up and running, until it is closed and the user is not exposed to the POSIX automaton.

Shutdown allows closing communication in only one way. After a shutdown, one of the peered sockets will only be allowed to send data and the other one will only be allowed to receive them.

Aborting a socket is a special way to close it, generally in response to an error. If an abort is requested on one socket, the peer closes it with an error message and does not wait till the pending data is handled.

In a TCP server-client connection between two remote devices if the peer socket is closed down abruptly (e.g. powered off) the peer TCP socket does not get any indication message. This is a normal behavior. But during a data transfer if there was an abnormal disconnection, WIP_CEV_ERROR occurs after a timeout period which indicates that the peer TCP socket is not reachable. However, if the peer TCP socket is closed normally then WIP_CEV_PEER_CLOSE is received in the event handler.

The TCP protocol uses a timeout mechanism to check the state of the TCP sockets in a TCP socket connection. According to this mechanism, to know the state of the peer TCP socket the data needs to be sent and wait for the acknowledgement within a specified time period. If the acknowledgement is not received within the specified time out period then the data is retransmitted. But if the time out occurs before receiving acknowledgement then it implies that the peer TCP socket is closed.

TCP Timeout Period = function (R, N)

Where,

R = Round trip time. This is the time for a TCP packet to go to the remote TCP socket and the time to receive the acknowledgement by the transmitter TCP socket. The typical round trip time is 1 second for GPRS.

N = Number of retransmission allowed before the time out happens.

The typical timeout period is 10 minutes depending on the network and also the peer TCP socket localization. Please note that the retransmission of the data to the peer TCP socket within the timeout period is managed by the Internet Library.

The maximum time between TCP retransmissions and the maximum number of retransmissions can be set using the options WIP_COPT_REXMT_MAX and WIP_COPT_REXMT_MAXCNT respectively. For more details about these options refer the section 6.4.2.2. Also there are other options WIP_NET_OPT_TCP_REXMT_MAX and WIP_NET_OPT_TCP_REXMT_MAXCNT for retransmissions that impact the entire stack. For details about these options refer the section 3.3.2.

It is possible to have a TCP client and TCP server sockets running at the same time in the same embedded module. In this scenario, when the connection is established between the TCP server and TCP client sockets, it is necessary to unmap the mapped socket in order to send/receive data on another socket. It is possible to use CMUX logical ports and can have an interface connection (like UART connection) for each socket for e.g. TCP client socket on one logical port and TCP server socket on another. In this case, it is not necessary to map or unmap the UART connections to send or receive the data from the socket.

2.5.5. UDP: UDP Socket

UDP sockets support the reading and writing of datagrams which are atomic data packets. However this does not guarantee that they arrive at the destination or that they arrive in order and are not duplicated. In addition to channel operations, they support a specific wip_UDPCreate() creation function. Since UDP does not work in a connected mode, there is no way for a socket to receive a WIP_CEV_PEER_CLOSE event. Write operations on UDP sockets are performed synchronously.

2.6. Options

Options are used for advanced channel control. First, the configuration of an open channel can be altered with setOpts() and read with getOpts(). Some options are mainly used at creation time (for example, while creating an account name for an anonymous FTP session). To handle such initialization-time options, for every foobarCreate() function, there is a dual foobarCreateOpts() function, which takes the same parameters as the former, plus a series of options settings. Finally, some protocols support special forms of read and write operations. In these cases, readOpts() and writeOpts() functions must be used instead of read() and write(); as expected, they take the same parameters as their counterparts without options, plus a series of options.

2.6.1. Option Series

In C language, a variable number of parameters can be passed to a function, for which types are not checked (because of the special "…" parameter). For the functions that accept options, we rely on a set of int constant values which identify channel options, prefixed with WIP_COPT_; for example, WIP_COPT_USERNAME, WIP_COPT_TRUNCATE and WIP_COPT_PORT. An option identifier is followed by its actual contents. For instance, WIP_COPT_USERNAME is followed by a const ascii* pointer which contains the user name as a string. The option name indicates the next data type to the function. It is possible for an option to take several parameters, or no parameter at all. Finally, C does not provide a way for a function accepting a variable number of parameters, to know when it has reached its last parameter. Therefore, a special option identifier WIP_COPT_END, which takes no value, indicates the end of the option series.

2.6.2. Example

Here is a simple write operation:

```
err = wip_write ( channel, buffer, buf_len );
```

A more elaborate writing, with some special settings would be as follows:

```
err = wip_writeOpts ( channel, buffer, buf_len,

                      WIP_COPT_DONTFRAG, true,

                      WIP_COPT_TTL, 5,

                      WIP_COPT_END );
```

The set of options accepted by an Opts functions depend on the underlying protocol of the channel. The function checks at runtime whether or not the options it receives are supported, and causes an ENOTSUPPORTED error when it receives an unsupported option. It is better to sort these options by channel type than by function. Hence, the API specification will hereafter be split by channel type rather than by function.

3. Initialization of the IP Connectivity Library

The IP connectivity library must be initialized by an application. During initialization, some parameters of the TCP/IP stack can be provided, such as the number of sockets and the memory used by network buffers. The default configuration should provide settings that are equivalent to the previous version of the TCP/IP stack.

The other modules of the IP connectivity library, the bearer manager and the socket communication layer, are also initialized by the functions described in the sections that follow.

3.1. Required Header File

The header file for the IP connectivity initialization is wip_net.h.

3.2. The wip_netInit Function

The wip_netInit function initializes the TCP/IP stack with a default configuration. This function or its variant wip_netInitOpts must be first called by the application before using any IP communication library service.

The memory is allocated for each predefined socket, network buffer etc. The memory required for the configuration can be calculated by, the size of the different elements such as number of sockets, socket buffers etc.

3.2.1. Prototype

```
s8 wip_netInit ( void );
```

3.2.2. Parameters

None

3.2.3. Returned Values

This function returns

- 0 if the TCP/IP stack has been successfully initialized
- In case of an error, the function returns a negative error code WIP_NET_ERR_NO_MEM only if an application is subscribed to adl_errSubscribe(), error code WM_EINVAL (-26) as an invalid argument value if task ID is greater than NETINT_MAX_THREADS, or otherwise, the module restarts

Note: *In a multitasking application, the wip_netinit API has to be called from each task that would need any IP communication library service, in order to reserve the associated execution context for each Internet Library operation.*

3.3. The wip_netInitOpts Function

The wip_netInitOpts function initializes the TCP/IP stack with some user defined options. This function or its variant wip_netInit must be called first by the application before using any IP communication library service.

The memory is allocated for each predefined socket, network buffer etc. The memory required for the configuration can be calculated by, the size of the different elements such as number of sockets, socket buffers etc. Refer section 3.2 for the size of different elements.

Since memory management is a delicate thing, it is recommended not to change default values to bigger ones. However, in case customer application requires such specific needs, it is recommended to subscribe to error management services through adl_errSubscribe() API : it will let the application catching memory related traps.

3.3.1. Prototype

```
s8 wip_netInitOpts ( int   opt,

                     ... );
```

3.3.2. Parameters

opt:

In: First option in the list of options.

...:

In: This function supports several parameters. These parameters are a list of options. The list of option names must be followed by option values. The list must be terminated by WIP_NET_OPT_END. The following options are currently defined:

Option	Value	Description	Default
WIP_NET_OPT_END	none	End of option list.	-
WIP_NET_OPT_TCP_REXMT_MAX	u32	The maximum time between TCP retransmissions.	64 seconds
WIP_NET_OPT_TCP_REXMT_MAXCNT	u32	The maximum number of retransmissions.	12
WIP_NET_OPT_IP_FORWARD	bool	Activate IP forwarding in NET.	FALSE
WIP_NET_OPT_IP_NAT_TO_TCP	s32	TCP flow timeout.	15 seconds
WIP_NET_OPT_IP_NAT_TO_TCP_FIN	s32	TCP FIN (no more data from sender) flow timeout.	2 seconds
WIP_NET_OPT_IP_NAT_TO_UDP	s32	UDP flow timeout.	5 seconds
WIP_NET_OPT_IP_NAT_TO_ICMP	s32	ICMP flow timeout.	2 seconds
WIP_NET_OPT_ARP_EXPIRE	s32	Expiration timeout of an ARP entry (deprecated option; no longer supported)	30 seconds

Option	Value	Description	Default
WIP_NET_OPT_TCP_KEEP_INIT	s32	Connection establishment timer value	75 seconds
WIP_NET_OPT_TCP_KEEP_IDLE	s32	Idle time before first probe	7200 seconds
WIP_NET_OPT_TCP_KEEP_INTVL	s32	Interval between probes when no response is received	50 seconds
WIP_NET_OPT_TCP_NOTIMEWAIT	bool	Enable/disable Time Wait state	0 (Time Wait state enabled)
WIP_NET_OPT_DHCPS_ADDR	wip_in_addr_t	Listening address of the DHCP server.	0.0.0.0 (any address)
WIP_NET_OPT_DHCPS_NB_ADDR	u32	Number of IP addresses of the range managed by the server.	0
WIP_NET_OPT_DHCPS_FIRST_ADDR	wip_in_addr_t	First IP address of the range managed by the server.	0.0.0.0
WIP_NET_OPT_DHCPS_SUBNET_MASK	wip_in_addr_t	Mask of the subnet managed by the server.	0.0.0.0
WIP_NET_OPT_DHCPS_LEASE	u32	Lease time for IP address (in seconds).	500
WIP_NET_OPT_DHCPS_MAX_LEASE	u32	Maximum lease time the server can provide (in seconds).	125 000
WIP_NET_OPT_DHCPS_GLOB_OPT[(1)]	wip_netDhcpOption_t *	Default configuration for clients that are not statically configured. The options are formatted as follows : Tag Value Length.	NULL
WIP_NET_OPT_DHCPS	bool	Activate DHCP server in NET. Valid configuration parameters must be set up before activation.	FALSE (disabled)

[(1)] Please refer to the DHCP options supported by the server section for more details

Note: *The range of values for the WIP_NET_OPT_TCP_REXMT_MAX option is the range of value coded on an u32 and the range of value for WIP_NET_OPT_TCP_REXMT_MAXCNT option is 0-12.*

Default values of the options WIP_NET_OPT_IP_NAT_TO_XXX are quiet small, since it was assumed that there is no connection which is left opened by the application in private network (like a telnet or a web server taking a long time to handle the request). If some applications in private network require such latencies, timeout values should be increased in NAT routers or applications should implement a “keep alive” feature in its own protocol.

3.3.3. Returned Values

The function returns

- 0 if the TCP/IP stack has been successfully initialized
- In case of an error, a error code as described below:

Error code	Description
WIP_NET_ERR_OPTION	Invalid option
WIP_NET_ERR_PARAM	Invalid option value
WIP_NET_ERR_NO_MEM	Memory allocation error
WM_EINVAL (-26)	Invalid argument value if task ID is greater than NETINT_MAX_THREADS

Note: *This function returns a negative error code WIP_NET_ERR_NO_MEM, only if an application is subscribed to adl_errSubscribe() otherwise, the embedded module restarts.*

3.4. The wip_netExit Function

The wip_netExit function terminates the TCP/IP stack and releases all resources (memory) allocated by wip_netInit or wip_netInitOpts.

Note: All bearers must be closed before calling that function.

3.4.1. Prototype

```
s8 wip_netExit ( void );
```

3.4.2. Parameters

None

3.4.3. Returned Values

The function always returns 0.

3.5. The wip_netSetOpts Function

The wip_netSetOpts function is used to set TCP/IP protocols options. See the table in the Parameters section for the available options.

3.5.1. Prototype

```
s8 wip_netSetOpts ( int   opt,

                     ... );
```

3.5.2. Parameters

opt:

In: First option in the list of options

...:

In: This function supports several parameters. These parameters are a list of options. The list of option names must be followed by option values. The list must be terminated by WIP_NET_OPT_END. The following options are currently defined:

Option	Value	Description
WIP_NET_OPT_IP_TTL	u8	Default TTL of outgoing datagrams. This option is a limit on the period of time or number of iterations or transmissions that a unit of data can experience before it should be discarded. The time to live (TTL) is an 8-bit field in the Internet Protocol (IP) header. It is the 9th octet of 20. The default value of this parameter is 64. Its value can be considered as an upper bound on the time that an IP datagram can exist in an internet system. The TTL field is set by the sender of the datagram, and reduced by every host on the route to its destination. If the TTL field reaches zero before the datagram arrives at its destination, then the datagram is discarded. This is used to avoid a situation in which an undelivered datagram keeps circulating in the network.

<table>
<tr><th>Option</th><th>Value</th><th>Description</th></tr>
<tr><td>WIP_NET_OPT_IP_TOS</td><td>u8</td><td>Default TOS of outgoing datagrams. The IP protocol provides a facility for the Internet layer to know about the various tradeoffs that should be made for a particular packet. This is required because paths through the Internet vary widely in terms of the quality of service provided. This facility is defined as the "Type of Service" facility, abbreviated as the "TOS facility". The TOS facility is one of the features of the Type of Service octet in the IP datagram header. The Type of Service octet consists of following three fields:
0 1 2 3 4 5 6 7
+---+----+----+----+----+----+----+----+
| | | |
| PRECEDENCE | TOS |MBZ|
| | | |
+---+----+----+----+----+----+----+----+
The first field is "PRECEDENCE". It is intended to denote the importance or priority of the datagram.
The second field is "TOS" which denotes how the network should maintain the tradeoffs between throughput, delay, reliability, and cost.
The last field is "MBZ" (Must Be Zero"), is currently unused and is set to 0.
The TOS field can have the following values:
1000 -- minimize delay
0100 -- maximize throughput
0010 -- maximize reliability
0001 -- minimize monetary cost
0000 -- normal service
For more information on this field please refer to RFC1349.
default:0</td></tr>
<tr><td>WIP_NET_OPT_IP_FRAG_TIMEO</td><td>u16</td><td>Time to live in seconds of incomplete fragments. When a datagram’s size is larger than the MTU (Maximum Transmission Unit) of the network, then the datagram is divided into smaller fragments. These divided fragments are sent separately. The “WIP_NET_OPT_IP_FRAG_TIMEO” option specifies the Time to live for these fragments.
default:30 seconds</td></tr>
</table>

Option	Value	Description
WIP_NET_OPT_TCP_MAXINITWIN	u16	Number of segments of initial TCP window. This option is used to specify the number of segments in the initial TCP window. A TCP window specifies the amount of outstanding (unacknowledged by the recipient) data a sender can send on a particular connection before it gets an acknowledgment back from the receiver. The primary reason for the window is congestion control. default:0
WIP_NET_OPT_TCP_MIN_MSS	u16	Default MSS for off-link connections. This option is used by the Internet Library internally. This parameter specifies the maximum size of TCP segment which would be sent. By default, the value of this parameter is set to 536. Hence Internet Library would not send any TCP segment having a length greater than 536 bytes without header.
WIP_NET_OPT_END	none	End of option list
WIP_NET_OPT_TCP_REXMT_MAX	u32	Sets the maximum time between TCP retransmissions. default:64 seconds
WIP_NET_OPT_TCP_REXMT_MAXCNT	u32	Sets the maximum number of retransmissions. default:12
WIP_NET_OPT_IP_FORWARD	bool	Activate IP forwarding in NET. default:FALSE
WIP_NET_OPT_IP_NAT_TO_TCP	s32	TCP flow timeout. default:15 seconds
WIP_NET_OPT_IP_NAT_TO_TCP_FIN	s32	TCP FIN (no more data from sender) flow timeout. default:2 seconds
WIP_NET_OPT_IP_NAT_TO_UDP	s32	UDP flow timeout. default:5 seconds
WIP_NET_OPT_IP_NAT_TO_ICMP	s32	ICMP
WIP_NET_OPT_ARP_EXPIRE	s32	Expiration timeout of an ARP entry (deprecated option; no longer supported) Default: 30 seconds
WIP_NET_OPT_TCP_KEEP_INIT	s32	Connection establishment timer value Default: 75 seconds
WIP_NET_OPT_TCP_KEEP_IDLE	s32	Idle time before first probe Default: 7200 seconds
WIP_NET_OPT_TCP_KEEP_INTVL	s32	Interval between probes when no response is received Default: 50 seconds
WIP_NET_OPT_TCP_NOTIMEWAIT	bool	Enable/disable Time Wait state Default: 0 (Time Wait state enabled)
WIP_NET_OPT_DHCPS_ADDR	wip_in_addr_t	Listening address of the DHCP server. Default : 0.0.0.0 (any address)

Option	Value	Description
WIP_NET_OPT_DHCPS_NB_ADDR	u32	Number of IP addresses of the range managed by the server. default: 0
WIP_NET_OPT_DHCPS_FIRST_ADDR	wip_in_addr_t	First IP address of the range managed by the server. default: 0.0.0.0
WIP_NET_OPT_DHCPS_SUBNET_MASK	wip_in_addr_t	Mask of the subnet managed by the server. default: 0.0.0.0
WIP_NET_OPT_DHCPS_LEASE	u32	Lease time for IP address (in seconds). Default: 500
WIP_NET_OPT_DHCPS_MAX_LEASE	u32	Maximum lease time the server can provide (in seconds). default: 125 000
WIP_NET_OPT_DHCPS_GLOB_OPT(1)	wip_netDhcpOption_t *	Default configuration for clients that are not statically configured. The options are formatted as follows : Tag Value Length. default : NULL
WIP_NET_OPT_DHCPS	bool	Activate DHCP server in NET. Valid configuration parameters must be set up before activation. default : FALSE (disabled)

(1) Please refer to the DHCP options supported by the server section for more details

3.5.3. Returned Values

The function returns

- 0 on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_NET_ERR_OPTION	Invalid option
WIP_NET_ERR_PARAM	Invalid option value

3.6. The wip_netGetOpts Function

The wip_netGetOpts function returns the current value of the TCP/IP protocols options that are passed in the argument list.

3.6.1. Prototype

```
s8 wip_netGetOpts ( int   opt,

                    ... );
```

3.6.2. Parameters

opt:

In: First option in the list of options

...:

In: This function supports several parameters. These parameters are a list of options. The list of option names must be followed by option values. The list must be terminated by WIP_NET_OPT_END. The following options are currently defined:

Option	Value	Description
WIP_NET_OPT_IP_TTL	u8	Default TTL of outgoing datagrams. This option is a limit on the period of time or number of iterations or transmissions that a unit of data can experience before it should be discarded. The time to live (TTL) is an 8-bit field in the Internet Protocol (IP) header. It is the 9th octet of 20. The default value of this parameter is 64. Its value can be considered as an upper bound on the time that an IP datagram can exist in an internet system. The TTL field is set by the sender of the datagram, and reduced by every host on the route to its destination. If the TTL field reaches zero before the datagram arrives at its destination, then the datagram is discarded. This is used to avoid a situation in which an undelivered datagram keeps circulating in the network.

<table>
<tr><th>Option</th><th>Value</th><th>Description</th></tr>
<tr><td>WIP_NET_OPT_IP_TOS</td><td>u8</td><td>Default TOS of outgoing datagrams. The IP protocol provides a facility for the Internet layer to know about the various tradeoffs that should be made for a particular packet. This is required because paths through the Internet vary widely in terms of the quality of service provided. This facility is defined as the "Type of Service" facility, abbreviated as the "TOS facility". The TOS facility is one of the features of the Type of Service octet in the IP datagram header. The Type of Service octet consists of following three fields:
0 1 2 3 4 5 6 7
+---+----+----+----+----+----+----+----+
| | | |
| PRECEDENCE | TOS |MBZ|
| | | |
+---+----+----+----+----+----+----+----+
The first field is "PRECEDENCE". It is intended to denote the importance or priority of the datagram.
The second field is "TOS" which denotes how the network should maintain the tradeoffs between throughput, delay, reliability, and cost.
The last field is "MBZ" (Must Be Zero"), is currently unused and is set to 0.
The TOS field can have the following values:
1000 -- minimize delay
0100 -- maximize throughput
0010 -- maximize reliability
0001 -- minimize monetary cost
0000 -- normal service
For more information on this field please refer to RFC1349.
default:0</td></tr>
<tr><td>WIP_NET_OPT_IP_FRAG_TIMEO</td><td>u16</td><td>Time to live in seconds of incomplete fragments. When a datagram’s size is larger than the MTU (Maximum Transmission Unit) of the network, then the datagram is divided into smaller fragments. These divided fragments are sent separately. The “WIP_NET_OPT_IP_FRAG_TIMEO” option specifies the Time to live for these fragments.
default:30 seconds</td></tr>
</table>

Option	Value	Description
WIP_NET_OPT_TCP_MAXINITWIN	u16	Number of segments of initial TCP window. This option is used to specify the number of segments in the initial TCP window. A TCP window specifies the amount of outstanding (unacknowledged by the recipient) data a sender can send on a particular connection before it gets an acknowledgment back from the receiver. The primary reason for the window is congestion control. default:0
WIP_NET_OPT_TCP_MIN_MSS	u16	Default MSS for off-link connections. This option is used by the Internet Library internally. This parameter specifies the maximum size of TCP segment which would be sent. By default, the value of this parameter is set to 536. Hence Internet Library would not send any TCP segment having a length greater than 536 bytes without header.
WIP_NET_OPT_END	none	End of option list
WIP_NET_OPT_TCP_REXMT_MAX	u32	Sets the maximum time between TCP retransmissions. default:64 seconds
WIP_NET_OPT_TCP_REXMT_MAXCNT	u32	Sets the maximum number of retransmissions. default:12
WIP_NET_OPT_IP_FORWARD	bool	Activate IP forwarding in NET. default:FALSE
WIP_NET_OPT_IP_NAT_TO_TCP	s32	TCP flow timeout. default:15 seconds
WIP_NET_OPT_IP_NAT_TO_TCP_FIN	s32	TCP FIN (no more data from sender) flow timeout. default:2 seconds
WIP_NET_OPT_IP_NAT_TO_UDP	s32	UDP flow timeout. default:5 seconds
WIP_NET_OPT_IP_NAT_TO_ICMP	s32	ICMP
WIP_NET_OPT_ARP_EXPIRE	s32	Expiration timeout of an ARP entry (deprecated option; no longer supported) Default: 30 seconds
WIP_NET_OPT_TCP_KEEP_INIT	s32	Connection establishment timer value Default: 75 seconds
WIP_NET_OPT_TCP_KEEP_IDLE	s32	Idle time before first probe Default: 7200 seconds
WIP_NET_OPT_TCP_KEEP_INTVL	s32	Interval between probes when no response is received Default: 50 seconds
WIP_NET_OPT_TCP_NOTIMEWAIT	bool	Enable/disable Time Wait state Default: 0 (Time Wait state enabled)
WIP_NET_OPT_DHCPS_NB_ADDR	u32	Number of IP addresses of the range managed by the server. default: 0

Option	Value	Description
WIP_NET_OPT_DHCPS_FIRST_ADDR	wip_in_addr_t	First IP address of the range managed by the server. default: 0.0.0.0
WIP_NET_OPT_DHCPS_SUBNET_MASK	wip_in_addr_t	Mask of the subnet managed by the server. default: 0.0.0.0
WIP_NET_OPT_DHCPS_LEASE	u32	Lease time for IP address (in seconds). Default: 500
WIP_NET_OPT_DHCPS_MAX_LEASE	u32	Maximum lease time the server can provide (in seconds). default: 125 000
WIP_NET_OPT_DHCPS_GLOB_OPT(1)	wip_netDhcpOption_t *	Default configuration for clients that are not statically configured. The options are formatted as follows : Tag Value Length. default : NULL
WIP_NET_OPT_DHCPS	bool	Activate DHCP server in NET. Valid configuration parameters must be set up before activation. default : FALSE (disabled)

(1) Please refer to the DHCP options supported by the server section for more details

3.6.3. Returned Values

The function returns

- 0 on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_NET_ERR_OPTION	Invalid option
WIP_NET_ERR_PARAM	Cannot get requested option value for internal reasons

4. IP Bearer Management

The IP bearer management API is used to initialize the TCP/IP network interfaces that work on top of the communication devices provided by ADL, including, but not limited to:

- UART
- GSM data
- GPRS
- Ethernet Bearer

The bearer management module is responsible for establishing the IP connectivity of the TCP/IP stack and configuring all the sub-layers of the network interface such as Ethernet, PPP, GSM data, and GPRS.

The API is asynchronous, all functions are non-blocking and events are reported through a callback function.

Some types of bearers (like UART, GSM) support a server mode where the bearer can wait for incoming connections. Authentication of the caller must be carried out by the application.

The API is not related to a specific type of bearer, and all bearer specific settings are handled by the Options mechanism. Support for new types of bearer devices (like USB, Bluetooth and so on) can be added by defining new options, without breaking the API.

Several network interfaces/bearers can be activated at the same time. IP routing is used for redirecting the data flow through the different interfaces.

The DNS resolver can also be configured by the bearer management module if the related information is provided by the server.

Note: *In a multitasking application, the bearer management must be done in the main application task context as it is not possible to manage bearers outside the main application task. But socket/session related operations can be done from other tasks contexts too.*

4.1. State Machine

The bearer management API exports a state machine to an application that is common for all bearer devices. The following states are defined:

State	Description
CLOSED	The IP bearer is closed; the device can be used by other software modules.
DISCONNECTED	The IP bearer is opened but not activated.
CONNECTING	Connection in progress.
CONNECTED	IP layer is configured; bearer can send and receive IP data
DISCONNECTING	Application has requested to disconnect the link; disconnection in progress.
PEER_DISCONNECTING	Peer has requested to disconnect the link or link-layer has detected a problem; disconnection in progress.
LISTENING	Waiting for connection requests/calls (server mode).
PEER_CONNECTING	Connection request from peer accepted by application, connection in progress.

The state transitions are shown in the figure below:

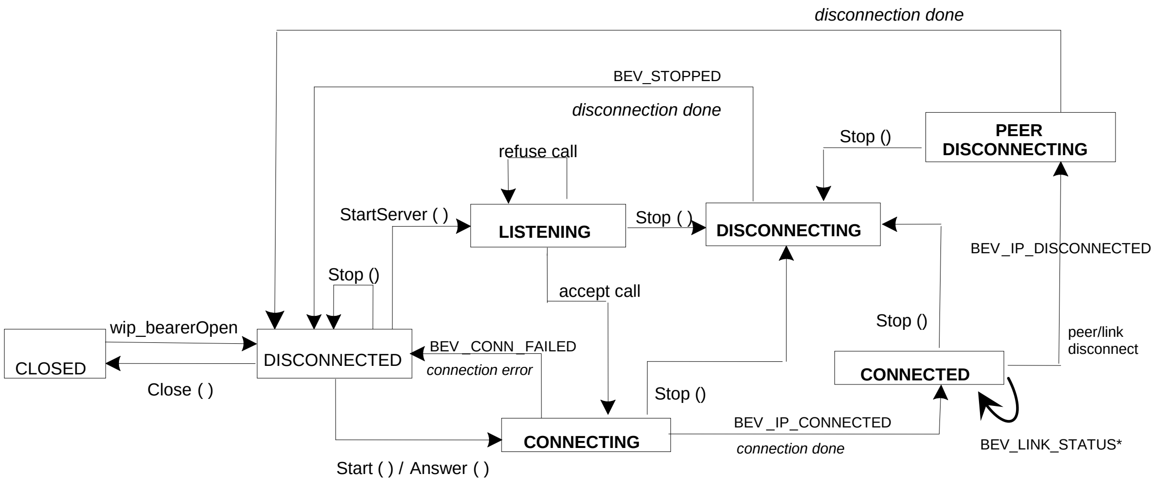


Figure 5. Bearer Management API State Diagram

The transitions are triggered by API function calls from the application or by the events reported by the link layer.

During some transitions, an event is reported to an application through the event notification callback function as follows:

Event	Description
WIP_BEV_CONN_FAILED	Connection failure, WIP_BOPT_ERROR returns the cause of the failure
WIP_BEV_IP_CONNECTED	IP communication ready
WIP_BEV_IP_DISCONNECTED	IP communication terminated, WIP_BOPT_ERROR returns the cause of the disconnection
WIP_BEV_STOPPED	Disconnection completed after wip_bearerStop was called
WIP_BEV_LINK_STATUS	Link status change indication (for Ethernet bearer only)
WIP_BEV_DRIVER	Driver event. Used by the driver to notify the application of a driver-specific event. The meaning of the event is driver dependant; this event is not interpreted by the Internet Library.

When the bearer is in the Listening state, an application can accept or refuse the connection request, through the server event notification callback as shown below:

Action	Description
Accept call	The notification callback has accepted the connection
Refuse call	The notification callback has refused the connection

4.2. Required Header File

The header file for the IP bearer management is wip_bearer.h.

4.3. IP Bearer Management Types

4.3.1. The wip_bearer_t Structure

The wip_bearer_t type is an opaque structure that stores a bearer handle.

4.3.2. The wip_bearerHandler_f Structure

The wip_bearerHandler_f type is an event handler callback.

Prototype

```
typedef void(*)  wip_bearerHandler_f ( wip_bearer_t   *br,
                                       s8   event,
                                       void   *context );
```

Parameters

br:

In: Bearer handle.

event:

In: Event name; the following events are currently defined:

Event	Description
WIP_BEV_CONN_FAILED	Connection failure, WIP_BOPT_ERROR returns the cause of the failure
WIP_BEV_IP_CONNECTED	IP communication ready
WIP_BEV_IP_DISCONNECTED	IP communication terminated, WIP_BOPT_ERROR returns the cause of the disconnection
WIP_BEV_STOPPED	Disconnection completed after wip_bearerStop was called
WIP_BEV_LINK_STATUS	Specifies link status. This event is passed on to Internet Application by Internet Library whenever ethernet link goes up, down, or there is link failure. Link status equals 1 when the ethernet link is down.
WIP_BEV_DRIVER	Driver event. Used by the driver to notify the application of a driver-specific event. The meaning of the event is driver dependant; this event is not interpreted by the Internet Library.
WIP_BEV_ME_UNREG	ME is not registered.
WIP_BEV_CTX_DEACT	Context is deactivated either from network or ME.

context:

In: Pointer to application context.

Returned Values

None

4.3.3. The wip_bearerServerHandler_f Structure

The wip_bearerServerHandler_f type is an event handler callback.

Prototype

```
typedef s8(*)  wip_bearerServerHandler_f (wip_bearer_t   br,
                                          wip_bearerServerEvent_t   *event,
                                          void *context);
```

Parameters

br:

In: Bearer handle.

event:

In: Event data of structure type wip_bearerServerEvent_t.

context:

In: Pointer to application context.

Returned Values

A positive value greater than zero is returned to accept the incoming connection, otherwise the call is rejected.

4.3.4. The wip_bearerType_e Type

The wip_bearerType_e enumeration stores the name and type of a bearer.

```
typedef enum {
  WIP_BEARER_NONE,
  WIP_BEARER_UART_PPP,
  WIP_BEARER_GSM_PPP,
  WIP_BEARER_GPRS
} wip_bearerType_e;
```

4.3.5. The wip_bearerInfo_t Structure

The wip_bearerInfo_t structure contains the name and type of a bearer.

```
typedef struct {
  ascii name[WIP_BEARER_NAME_MAX];
  wip_bearerType_e  type;
} wip_bearerInfo_t;
```

4.3.6. The wip_bearerDrvOption_t Structure

This structure is used for passing driver options. It is used internaly by API functions.

```
typedef struct {
  s32   optname;
  void *optval;
  s32   optlen;
  s32   ret;
} wip_bearerDrvOption_t;
```

Parameters

optname:

In: name of option, specific to the driver.

optval:

In: option value.

optlen:

In: length of option value.

ret:

Out: result code

4.3.7. The wip_bearerServerEvent_t Structure

This structure is used for passing server event information.

```
typedef struct {
  S8   kind;
  wip_bearerServerEvent_t::wip_bearerServerEventContent_t  content;
  union   wip_bearerServerEventContent_t;
} wip_bearerServerEvent_t;
```

Parameters

kind:

In: Event name. This contains the following event names:

Kind	Description
WIP_BEV_DIAL_CALL	Signals an incoming call. When this event occurs the structure dial_call should be used to extract the parameters. This structure contains the phone number of caller. The callback function must return a positive value to accept the call.
WIP_BEV_PPP_AUTH_PEER	Signals a PPP peer authentication request. When this event occurs the structure ppp_auth should be used to extract the parameters. This structure contains the user name provided by the peer. The callback function must return a positive value if the user name is correct, and fill the secret buffer with the secret data (password) associated with the user. The bearer will then check if the secret data given by the peer is correct.

`phonenb:`

In: Phone number of the caller.

`user:`

In: User name given by caller.

`userlen:`

In: Length of user name.

`secret:`

In: Pointer to a buffer to be filled with the secret data of the user.

`secretlen:`

In: Initialized with the maximum allowed length of the secret, must contains the length of the secret after the call.

4.3.8. The wip_ifindex_t Structure

The wip_ifindex_t type is an opaque structure that stores an interface index. Interface indexes are used by the TCP/IP stack to reference a network interface.

4.4. The wip_bearerOpen Function

The wip_bearerOpen function attaches a bearer device to a network interface. Depending on the type of bearer, the network interface will implement PPP or will work in packet mode. The bearer is identified by a string. The caller must specify an event handler callback and a context to process the bearer-related asynchronous events. See the DUAL PDP Support section for more details.

The bearer is initialized with a default configuration that can be changed by wip_bearerSetOpts. The bearer and its associated network must be activated by wip_bearerStart or wip_bearerStartServer in order to enable IP communication.

4.4.1. Prototype

```
s8 wip_bearerOpen ( wip_bearer_t   *br,
                    const ascii   *device,
                    wip_bearerHandler_f   brHdlr,
                    void   *context );
```

4.4.2. Parameters

br:

Out: Filled with bearer handle if the open function was successful.

context:

In: Pointer to application defined context that is passed to the event handler callback.

device:

In: Bearer name, the currently supported devices are listed below:

Device	Description
UART1	UART 1, PPP mode
UART1x	DLC 'x' on UART 1, 'x' from 1 to 4, PPP mode
UART2	UART 2, PPP mode
UART2x	DLC 'x' on UART 2, 'x' from 1 to 4, PPP mode
GSM	GSM data, PPP mode
GPRS	GPRS, packet mode
GPRSx	GPRS, packet mode for DUAL PDP Context Support *Note:* *See the the DUAL PDP Support section for more details about DUAL PDP context over GPRS bearer.*

Note: *If one physical UART is multiplexed into DLCs (DLC1, DLC2, DLC3, DLC4), only one among these DLCs can be used for PPP over session.*

brHdlr:

In: Event handler callback, the function has the following prototype:

```
typedef void (*wip_bearerHandler_f) ( wip_bearer_t   br,
                                      s8   event,
                                      void   *context );
```

br:

In: Bearer handle

event:

In: Event name, the following events are currently defined:

Event	Description
WIP_BEV_CONN_FAILED	Connection failure, WIP_BOPT_ERROR returns the cause of the failure
WIP_BEV_IP_CONNECTED	IP communication ready
WIP_BEV_IP_DISCONNECTED	IP communication terminated, WIP_BOPT_ERROR returns the cause of the disconnection
WIP_BEV_STOPPED	Disconnection completed after wip_bearerStop was called
WIP_BEV_LINK_STATUS	Specifies link status. This event is passed on to Internet Application by Internet Library whenever ethernet link goes up, down, or there is link failure. Link status equals 1 when the ethernet link is down.
WIP_BEV_DRIVER	Driver event. Used by the driver to notify the application of a driver-specific event. The meaning of the event is driver dependant; this event is not interpreted by the Internet Library.
WIP_BEV_ME_UNREG	ME is not registered
WIP_BEV_CTX_DEACT	Context is deactivated either from network or ME

context:

In: Pointer to application context

Returned Values:

None

4.4.3. Returned Values

The function returns

- 0 on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_BERR_NO_DEV	The device does not exist
WIP_BERR_ALREADY	The device is already opened
WIP_BERR_NO_IF	The network interface is not available
WIP_BERR_NO_HDL	No free handle or Max GPRS PDP context reached.
WIP_BERR_BAD_STATE	The corresponding task ID is invalid.
WIP_BERR_NO_MEM	Internet Library is unable to allocate memory for bearer structure.

Note: *WIP_BEV_DIAL_CALL and WIP_BEV_PPP_AUTH_PEER are to be used only in handler installed by wip_bearerStartServer; they have no meaning outside that context.*

4.5. The wip_bearerClose Function

The wip_bearerClose function detaches the bearer from the network interface and releases all associated resources. If the bearer is not stopped the underlying connection is terminated but no event is generated. After the call, the associated TCP/IP network is closed and it will be available for another bearer association.

4.5.1. Prototype

```
s8 wip_bearerClose ( wip_bearer_t   br );
```

4.5.2. Parameters

`br:`

In: Bearer handle

4.5.3. Returned Values

The function returns

- 0 on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_BERR_BAD_HDL	Invalid handle
WIP_BERR_BAD_STATE	Bearer was not stopped before closing

4.6. The wip_bearerSetOpts Function

The wip_bearerSetOpts function sets configuration options of a bearer.

Note: It should be called before wip_bearerStart to setup the connection parameters.

4.6.1. Prototype

```
s8 wip_bearerSetOpts ( wip_bearer_t   br,

                       int   opt,

                       ... );
```

4.6.2. Parameters

br:

In: Bearer handle

opt:

In: First option in the list of options

...:

In: List of option names followed by option values. The list must be terminated by WIP_BOPT_END.

The following options are currently defined:

Option	Value	Description
WIP_BOPT_NAME	ascii	Name of bearer device (get only)
WIP_BOPT_TYPE	wip_bearerType_e	Type of bearer (get only)
WIP_BOPT_IFINDEX	wip_ifindex_t	Index of network interface (get only)
WIP_BOPT_ERROR	s8	Error code indicating the cause of the disconnection (get only) default:0
WIP_BOPT_RESTART	bool	Automatically restart server after connection is terminated default:FALSE
WIP_BOPT_END	none	End of option list
WIP_BOPT_LOGIN	ascii	Username default:0
WIP_BOPT_PASSWORD	ascii	Password default:0
Dialing Options		
WIP_BOPT_DIAL_PHONENB	ascii	Phone number

Option	Value	Description
WIP_BOPT_DIAL_RINGCOUNT	u16	Number of rings to wait before sending the WIP_BEV_DIAL_CALL event default:3 (only used for WIP_BEARER_GSM_PPP)
WIP_BOPT_DIAL_MSNULLMODEM	bool	Enable MS-Windows null-modem protocol ("CLIENT"/"SERVER" handshake) default:TRUE (only used for WIP_BEARER_UART_PPP)
WIP_BOPT_DIAL_SPEED	u32	Speed (in bits per second) of the connection (get only) PPP Options
WIP_BOPT_PPP_PAP	bool	Allow PAP authentication default:TRUE
WIP_BOPT_PPP_CHAP	bool	Allow CHAP authentication default:TRUE
WIP_BOPT_PPP_MSCHAP1	bool	Allow MSCHAPv1 authentication default:TRUE
WIP_BOPT_PPP_MSCHAP2	bool	Allow MSCHAPv2 authentication default:TRUE
WIP_BOPT_PPP_ECHO	bool	Send LCP echo requests to check if peer is alive default:TRUE for WIP_BEARER_ETHER and FALSE for WIP_BEARER_GSM_PPP
GPRS options		
WIP_BOPT_GPRS_APN	ascii	Address of GGSN default:0 (only used for WIP_BEARER_GPRS)
WIP_BOPT_GPRS_CID	u8	Cid of the PDP context default:1 (only used for WIP_BEARER_GPRS)
WIP_BOPT_GPRS_HEADERCOMP	bool	Enable PDP header compression default:FALSE (only used for WIP_BEARER_GPRS)
WIP_BOPT_GPRS_DATACOMP	bool	Enable PDP data compression default:FALSE (only used for WIP_BEARER_GPRS)
WIP_BOPT_GPRS_ERROR_FORWARD	adl_port_e	To forward <GPRS errors> to external application on specified port default = ADL_PORT_NONE (no forwarding)
IP Options		
WIP_BOPT_IP_ADDR	wip_in_addr_t	Local IP address default:0
WIP_BOPT_IP_DST_ADDR	wip_in_addr_t	Destination IP address default:0
WIP_BOPT_IP_DNS1	wip_in_addr_t	Address of primary DNS server default:0
WIP_BOPT_IP_DNS2	wip_in_addr_t	Address of secondary DNS server default:0

Option	Value	Description
WIP_BOPT_IP_SETDNS	bool	Configure DNS resolver when connection is established default:TRUE
WIP_BOPT_IP_SETGW	bool	Set interface as default gateway when connection is established default:TRUE
WIP_BOPT_EXTNAT	bool	Enable the NAT for the interface (bearer).By default (FALSE) all the interfaces are private. Setting this option to TRUE will mark the interface to public.

Note: *The options WIP_BOPT_IP_ADDR, WIP_BOPT_IP_DST_ADDR, WIP_BOPT_IP_DNS1 and WIP_BOPT_IP_DNS2 can be read only after the bearer connection is established successfully. If an attempt is made to read the options value before the bearer connection is established successfully incorrect IP address will be received.*

4.6.3. Returned Values

The function returns

- 0 on success
- In case of an error, an error code as described below:

Error Code	Description
WIP_BERR_BAD_HDL	Invalid handle
WIP_BERR_OPTION	Invalid option
WIP_BERR_PARAM	Invalid option value
WIP_BERR_BAD_STATE	Set option not allowed in the current Bearer state.

In case of an error, the following GPRS errors can also appear as listed in the table below:

GPRS Error Code	Meaning	Resulting from the following commands
103	Incorrect MS identity.(#3)	+CGATT
132	Service option not supported (#32)	+CGACT +CGDATA ATD*99
133	Requested service option not subscribed (#33)	+CGACT +CGDATA ATD*99
134	Service option temporarily out of order (#26, #34, #38)	+CGACT +CGDATA ATD*99
148	Unspecified GPRS error	All GPRS commands
149	PDP authentication failure (#29)	+CGACT +CGDATA ATD*99
150	Invalid mobile class	+CGCLASS +CGATT

4.7. The wip_bearerGetOpts Function

The wip_bearerGetOpts function retrieves configuration options and status variables of a bearer. It can be called after the connection is established to get the configuration parameters given by the peer (IP and DNS server addresses, link specific parameters, and so on).

4.7.1. Prototype

```
s8 wip_bearerGetOpts ( wip_bearer_t   br,
                       int   opt,
                       ... );
```

4.7.2. Parameters

br:

In: Bearer handle

opt:

In: First option in the list of options

...:

In/Out: For the list of options followed by pointers to option values, see the wip_bearerSetOpts Function options section.

4.7.3. Returned Values

The function returns:

- 0 on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_BERR_BAD_HDL	Invalid handle
WIP_BERR_OPTION	Invalid option

4.8. The wip_bearerStart Function

The wip_bearerStart function establishes the bearer connection. Depending on the type of bearer the following operations are made:

UART Device

- start the window's null-modem protocol handshake (if enabled)
- start PPP in client mode, IP connectivity is established by the PPP interface

GSM Device

- setup GSM data connection
- start PPP in client mode, IP connectivity is established by the PPP interface

GPRS Device

- set up GPRS connection
- configure IP address and DNS resolver with information returned by GGSN and enable IP communication on the interface

Note: There is no mechanism that deals with actions conflicts on bearer management application side (ADL or AT parser in firmware). E.g. ATH from external terminal stops the bearer link for GSM/GPRS bearer. ATDxxx; will stop the GPRS bearer etc.

4.8.1. Prototype

```
s8 wip_bearerStart ( wip_bearer_t   br );
```

4.8.2. Parameters

br:

In: Bearer handle

4.8.3. Events

After calling wip_bearerStart, the following events can be received:

Event	Description
WIP_BEV_IP_CONNECTED	The connection is completed
WIP_BEV_IP_DISCONNECTED	Peer has disconnected the link, or a link failure has been detected, call wip_bearerGetOpts with WIP_BOPT_ERROR option to get the cause of disconnection
WIP_BEV_IP_DISCONNECTED	The connection has failed to complete, call wip_bearerGetOpts with WIP_BOPT_ERROR option to get the cause of failure

After a connection failure, the WIP_BOPT_ERROR option can returns one of the following error codes:

Error	Description
WIP_BERR_LINE_BUSY	Line busy
WIP_BERR_NO_ANSWER	No answer
WIP_BERR_NO_CARRIER	No carrier
WIP_BERR_NO_SIM	No SIM card inserted
WIP_BERR_PIN_NOT_READY	PIN code not entered
WIP_BERR_GPRS_FAILED	GPRS setup failure
WIP_BERR_PPP_LCP_FAILED	LCP negotiation failure
WIP_BERR_PPP_AUTH_FAILED	PPP authentication failure
WIP_BERR_PPP_IPCP_FAILED	IPCP negotiation failure
WIP_BERR_PPP_LINK_FAILED	PPP peer not responding to echo requests
WIP_BERR_PPP_TERM_REQ	PPP session terminated by peer
WIP_BERR_CALL_REFUSED	Incoming call refused

4.8.4. Returned Values

The function returns

- 0 on success (bearer is connected)
- WIP_BERR_OK_INPROGRESS when bearer is connecting (an event will be sent after completion)
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_BERR_BAD_HDL	Invalid handle
WIP_BERR_BAD_STATE	The bearer is not stopped or an another GPRS bearer is in transition progress (connecting, disconnecting or peer disconnecting) *Note: See the DUAL PDP Support section for more details about DUAL PDP context over GPRS bearer.*
WIP_BERR_DEV	Error from link layer initialization

4.9. The wip_bearerAnswer Function

The wip_bearerAnswer function is used to answer an incoming phone call and start the bearer in the passive (server) mode. This function is only supported by the GSM bearer.

4.9.1. Prototype

```
s8 wip_bearerAnswer ( wip_bearer_t   br,
                      wip_bearerServerHandler_f   brSrvHdlr,
                      void   *context );
```

4.9.2. Parameters

br:

In: Bearer handle

brSrvHdlr:

In: Server event handler callback. The brSrvHdlr can only handle WIP_BEV_PPP_AUTH_PEER kind of event. Refer section 4.10.2 for details on the call back function prototype.

context:

In: Pointer to application context

4.9.3. Events

See the wip_bearerStart event list.

4.9.4. Returned Values

The function returns:

- 0 on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_BERR_BAD_HDL	Invalid handle
WIP_BERR_BAD_STATE	Bearer is not stopped
WIP_BERR_NOT_SUPPORTED	Not a GSM bearer
WIP_BERR_DEV	Error from link layer initialization

4.10. The wip_bearerStartServer Function

The wip_bearerStartServer function starts the bearer in passive (server) mode. The bearer waits for incoming connection requests. The WIP_BEV_DIAL_CALL event is generated when a call is received, the server handler callback can accept or refuse the call. If the call is accepted, the protocol layers configuration is started.

UART Device

- wait for incoming PPP connection on the UART port (WIP_BEV_PPP_AUTH_PEER is received)

GSM Device

- first wait for incoming GSM call in data mode (WIP_BEV_DIAL_CALL is received => accepting the call will establish the radio link).
- then wait for incoming PPP connection on that radio link (WIP_BEV_PPP_AUTH_PEER is received)

GPRS Device

- this function is not supported by the GPRS bearer

4.10.1. Prototype

```
s8 wip_bearerStartServer ( wip_bearer_t   br,

                           wip_bearerServerHandler_f   brSrvHdlr,

                           void   *context );
```

4.10.2. Parameters

br:

In: Bearer handle

brSrvHdlr:

In: Server event handler callback, the function has the following prototype:

```
typedef s8 (*wip_bearerServerHandler_f) ( wip_bearer_t   br

                                          wip_bearerServerEvent_t   *event,

                                          void   *context );
```

event:

In: Event data, the structure bearerServerEvent_t has the following definition:

```
typedef struct {
  s8 kind;
  union wip_bearerServerEventContent_t {
    struct wip_bearerServerEventContentDialCall_t {
```

```
      ascii  *phonenb;
    } dial_call;
    struct wip_bearerServerEventContentPppAuth_t {
      ascii *user;
      int userlen;
      ascii *secret;
      int secretlen;
    } ppp_auth;
  } content;
} wip_bearerServerEvent_t;
```

The structure members are described below.

kind:

In: Event name. This contains the following event names:

Kind	Description
WIP_BEV_DIAL_CALL	Signals an incoming call. When this event occurs the structure dial_call should be used to extract the parameters. This structure contains the phone number of caller. The callback function must return a positive value to accept the call.
WIP_BEV_PPP_AUTH_PEER	Signals a PPP peer authentication request. When this event occurs the structure ppp_auth should be used to extract the parameters. This structure contains the user name provided by the peer. The callback function must return a positive value if the user name is correct, and fill the secret buffer with the secret data (password) associated with the user. The bearer will then check if the secret data given by the peer is correct.

phonenb:

Phone number of the caller

user:

User name given by caller

userlen:

Length of user name

secret:

Pointer to a buffer to be filled with the secret data of the user

secretlen:

Initialized with the maximum allowed length of the secret, must contains the length of the secret after the call.

context:

In: Pointer to application context.

Returned Values:

A positive value greater than zero is returned to accept the incoming connection, otherwise the call is rejected.

4.10.3. Events

See the wip_bearerStart event list.

4.10.4. Returned Values

The function returns

- 0 on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_BERR_BAD_HDL	Invalid handle
WIP_BERR_BAD_STATE	The bearer is not stopped
WIP_BERR_NOT_SUPPORTED	Bearer does not support passive mode
WIP_BERR_DEV	Error from link layer initialization

4.11. The wip_bearerStop Function

The wip_bearerStop function terminates connection on a bearer. If the connection is still in progress, the connection is aborted. The following operations are made:

- the network interface is closed, and in case of PPP interface, the PPP connection is gradually stopped
- the link connection (GSM, GPRS) is terminated
- the WIP_BEV_STOPPED event is sent after all layers are properly shut down
- If the bearer is already stopped, then the function has no effect.
- A WIP_BERR_BAD_STATE error code is returned if another GPRS bearer is in transition progress (connecting, disconnecting or peer disconnecting).

4.11.1. Prototype

```
s8 wip_bearerStop ( wip_bearer_t   br );
```

4.11.2. Parameters

br:

In: Bearer handle

4.11.3. Events

After calling wip_bearerStop, the following events can be received:

Event	Description
WIP_BEV_STOPPED	The bearer is disconnected

4.11.4. Returned Values

This function returns

- 0 on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_BERR_OK_INPROGRESS	Disconnection in progress, a WIP_BEV_STOPPED event will be sent after completion
WIP_BERR_BAD_HDL	Invalid handle
WIP_BERR_BAD_STATE	Another GPRS bearer is in transition progress (connecting, disconnecting or peer disconnecting) *Note: See the DUAL PDP Support section for more details about DUAL PDP context over GPRS bearer.*

4.12. The wip_bearerGetList Function

The wip_bearerGetList function returns the list of all available bearers. This function always returns the same values for a given platform.

4.12.1. Prototype

```
wip_bearerInfo_t *wip_bearerGetList ( void );
```

4.12.2. Parameters

None

4.12.3. Returned Values

The function returns

- an array of wip_bearerInfo_t on success
- NULL pointer is returned on error. The end of the array is indicated by an entry with WIP_BEARER_NONE type and "" name. The memory used by the array is allocated dynamically and must be freed by calling wip_bearerFreeList.

Note: *The list of available bearers is not dynamically updated by other ADL calls. E.g. if customer application start a GSM call independently of Internet Library API, then wip_bearerGetList will still describe GSM bearer as available even if it is not the case at the moment. Availability of a bearer is only tested when the bearer is started by calling wip_bearerStart, wip_bearerAnswer or wip_bearerStartServer.*

4.13. The wip_bearerFreeList Function

The wip_bearerFreeList function frees the memory previously allocated by wip_bearerGetList.

4.13.1. Prototype

```
void wip_bearerFreeList ( wip_bearerInfo_t   *binfo );
```

4.13.2. Parameters

binfo:

In: Pointer that was returned by wip_bearerGetlist

4.13.3. Returned Values

None

4.14. The wip_bearerGetDrvOption Function

This function is used to get driver specific options. each driver can define its own set of functions.

Note: This function can be called when the bearer is not started.

4.14.1. Prototype

```
s8 wip_bearerGetDrvOption( wip_bearer_t br, s32 optname,
                           void *optval, s32 *optlen,
                           s32 *ret);
```

4.14.2. Parameters

br:

In: bearer handle

optname:

In: name of option, specific to the driver.

optval:

Out: buffer filled with option value.

optlen:

In: length of buffer.

Out: length of returned option value

ret:

Out: result code

4.14.3. Returned Values

The function returns 0 upon success and the application checks the result code. If unsuccessful, the function returns a negative error code.

4.15. The wip_bearerSetDrvOption Function

This function is used to set driver specific options. Each driver can define its own set of functions.

Note: This function can be called when the driver is not active.

4.15.1. Prototype

```
s8 wip_bearerSetDrvOption( wip_bearer_t br, s32 optname,
                           const void *optval, s32 optlen,
                           s32 *ret);
```

4.15.2. Parameters

br:

In: bearer handle

optname:

In: name of option, specific to the driver.

optval:

In: option value.

optlen:

In: length of option value.

ret:

Out: result code

4.15.3. Return value

The function returns 0 upon success and the application checks the result code. If unsuccessful, the function returns a negative error code.

4.16. IP Routing Management

The IP routing table is used to store the routes and to determine where data packets travelling over an IP network will be directed. Thus, it shows information about the network topology. Additional routes may be added in this table to select which bearer will be used to send IP packets to specific IP addresses. Note that for a specific bearer, a route is created automatically at bearer start according to the IP address obtained by the bearer.

There are 2 new functions which permit to change the IP routing table.

4.16.1. The wip_ipRouteAdd Function

The wip_ipRouteAdd function adds a static route to the specified host or network. The specified gateway must be directly attached to one of the local interfaces. A zero value for address and netmask specifies a default route.

Prototype

```
s8 wip_ipRouteAdd( wip_in_addr_t   addr,

                   wip_in_addr_t   mask,

                   wip_in_addr_t   gateway );
```

Parameters

addr:

In: Host or destination address.

mask:

In: Sub-network part of address mask.

gateway:

In: Address of gateway.

Returned Values

The function returns

- 0 if the addition of route was successful
- Negative error code in case of an error as described below:

Error Code	Description
WM_EINVAL	A parameter is not valid.
WM_ENOSPC	If the routing table is full.

4.16.2. The wip_ipRouteDel Function

The wip_ipRouteDel removes the route associated with the specified host or network.

Prototype

```
s8 wip_ipRouteDel( wip_in_addr_t dest);
```

Parameters

dest:

In: Host or network destination address of the route.

Returned Values

The function returns

- 0 on success
- In case of an error, negative error code as described below:

Error Code	Description
WM_EINVAL	If destination address specified cannot be found in the routing table.

4.17. NAT feature

4.17.1. Introduction to NAT

NAT (Network Address Translation) is a network service which is required to modify the network address information in the datagram packet headers while in transit across a traffic routing device. This is mainly for the purpose of remapping a given address space into another.

Network Address Port Translation (NAPT) also known as “IP masquerading”, is a technique to hide an entire address space (private network addresses) behind a single IP address in another address space (public addresses). This feature is now added as NET in Firmware which allows a Sierra Wireless embedded module with the application to work like a gateway.

The following example will explain the need of NAT service:

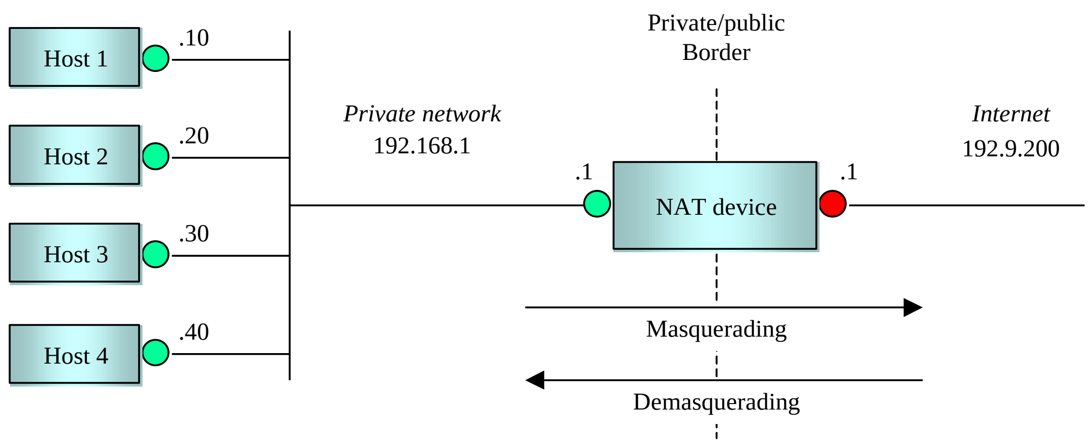


Figure 6. Simple NAT device example

As you can see in Figure 6, the simplest example of a NAT device (e.g. an IP router) is a host with only 2 interfaces: one directly connected to an Ethernet or WIFI LAN, and another one for a connection to an ISP (e.g. using PPP on GSM or GPRS). The NAT router will then allow hosts on LAN to gain connectivity to the internet.

It is a fact that the Internet's transport protocol IPv4 does not provide enough unique addresses for all the new hosts on the internet. A solution is to convert private internal addresses to official addresses when crossing the border from private network to the internet as shown in the Figure 6. This works because the number of hosts that communicate over the internet at a given time is considerably lower than the total number of hosts potentially connected.

This technique greatly helps to save address space, because only hosts currently communicating will dynamically use an official address assigned by a NAT router.

NAPT allows "n to 1" NAT translation, meaning that a single IP address, valid across the internet, can be used by several hosts behind NAT on some private network. However basic NAT may be used if a pool of valid external IP (public IP) addresses are available (not just one), allowing some form of "n to m" NAT.

4.17.2. NAT Tables

NAT tables are used to maintain information about the IP datagrams that pass through the NAT device. Each entry in the table includes a flow identifier. The NAT table contains runtime data which can change depending on flow creation and deletion frequency. For current implementation, NAT table is statically configured to handle up to 64 flows.

There are options WIP_NET_OPT_IP_NAT_TO_XXX to configure the timeout of the NAT entries in the NAT table. Please refer to the section 3.3.2 or 3.5.2 for details about these options.

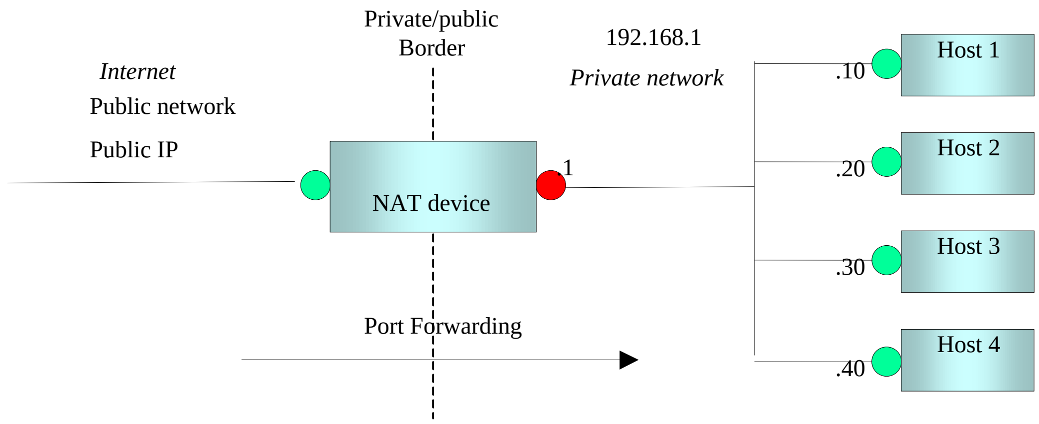
4.17.3. How to use NAT feature

The NAT feature will work only if the IP forwarding in NET is activated with the option WIP_NET_OPT_IP_FORWARD using wip_netSetOpts or the wip_netInitOpts functions. For the details about this option, please refer to the section 3.3.2 or 3.5.2.

An option WIP_BOPT_EXTNAT is provided to allow the application to set a public bearer using wip_bearerSetOpts function. This option can also be used to retrieve current NAT state of the specified bearer with the wip_bearerGetOpts function (public or private bearer). For the details about this option, please refer to section 4.6.2.

4.18. Port Forwarding Management

Port Forwarding is the technique of forwarding a TCP or UDP packet from a public network interface to a predetermined private network port and address through a Network Address Translator (Nat/Gateway). This Feature permits an access from public network to machine on private network.



Public interface receives packets on (Public address: Public Port) and forwards to (Private address: Private Port)

Port forwarding table can manage until 64 entries. There are 2 new functions which permit to add or delete entry in the Port Forwarding table.

For GPRS services, Network operator must provide a Public IP to gateway to be available to receive incoming public packet.

NAT feature must be activated (see chapter NAT feature).

4.18.1. The wip_ipFwdEntryAdd Function

The wip_ipFwdEntryAdd function adds a static rule in Port Forwarding table associated with the public address, port and protocol.

Prototype

```
s8 wip_ipFwdEntryAdd(wip_in_addr_t dest_ip,
                     u16 dest_port,
                     wip_in_addr_t gateway_ip,
                     u16 gateway_port,
                     u8 protocol );
```

Parameters

dest_ip:

In: Destination address.

`dest_ port:`

In: Destination port.

`gateway _ip:`

In: Public address.

`gateway _ port:`

In: Public port.

`protocol:`

In: Protocol.(IPPROTO_TCP or IPPROTO_UDP)

Returned Values

The function returns

- 0 if the addition of route was successful
- Negative error code in case of an error as described below:

Error Code	Description
WIP_NET_ERR_EADDRINUSE	Public Address and port already in use.
WIP_NET_ERR_PARAM	A parameter is not valid.
WIP_NET_ERR_ENOSPC	The static port forwarding table is full.

4.18.2. The wip_ipFwdEntryDel Function

The wip_ipFwdEntryDel removes the static rule in Port Forwarding table associated with the public address, port and protocol.

Prototype

```
s8 wip_ipFwdEntryDel(wip_in_addr_t gateway_ip,
                     u16 gateway_port,
                     u8 protocol );
```

Parameters

`gateway _ip:`

In: Public address.

`gateway _ port:`

In: Public port.

`protocol:`

In: Protocol.(IPPROTO_TCP or IPPROTO_UDP)

Returned Values

The function returns

- 0 on success
- In case of an error, negative error code as described below:

Error Code	Description
WIP_NET_ERR_ENOFOUND	Entry not found
WIP_NET_ERR_PARAM	A parameter is not valid.

4.19. DHCP server feature

4.19.1. Introduction to DHCP server feature

The Dynamic Host Configuration Protocol (DHCP) is a computer networking protocol allowing network devices (DHCP clients) to join an IP-based network without having a pre-configured IP address.

DHCP is built on a client-server model, where designated DHCP server hosts allocate network addresses and deliver configuration parameters to dynamically configured hosts. Devices running DHCP client software can then automatically retrieve these settings from DHCP servers as needed.

DHCP environments require a DHCP server set up with the appropriate configuration parameters for the given network. Key DHCP parameters include the range or "pool" of available IP addresses, the correct subnet masks, plus gateway and name server addresses.

The DHCP server :

- implements UDP as its transport protocol. DHCP messages from the server to a client are sent to the 'DHCP client' port (68).
- allocates a unique IP address to each of its clients. It can also be used to provide some configuration parameters to complete the client initialisation.
- guarantees that no specific network address is used by more than one DHCP client at a time.
- retains DHCP client configuration all the way through DHCP client reboot (assigning the same configuration parameters to the client).

The DHCP server can receive one of the five following client messages: DHCPDISCOVER, DHCPREQUEST, DHCPDECLINE, DHCPRELEASE and DHCPINFORM. The server will respond with one of the following three message types :

- DHCPOFFER: offers of some configuration parameters in response to DHCPDISCOVER
- DHCPACK: message with configuration parameters, including the granted network address
- DHCPNAK: indicates that the client's notion of the network address is incorrect

RFC 2131 gives the use of the fields and options in a DHCP message by a server.

Note: *The DHCP server feature has been locked as a commercial feature named "internet plug-in" up to and including firmware release 7.44. If the feature is not enabled due to using a firmware prior to 7.45, you can refer to the Firmware AT Commands Interface Manual (specifically the AT+WCFM command), and contact your Sierra Wireless distributor or sales point for further details.*

4.19.2. How to use DHCP server feature

DHCP server provides an application interface for dynamically distributing the IP address to the destination host, using Sierra Wireless TCP/IP implementation (Internet Library).

The DHCP server exists independently from the bearer (not limited to Ethernet) as any Ethernet like network may need a DHCP server. However it must be associated to an interface (listening interface).

The feature will work only if LAN bearer auto-configuration (DHCP client mode) is disabled with the option WIP_BOPT_IP_DHCP set to FALSE.

Specific options WIP_NET_OPT_DHCPS_XXX are provided to configure the server at initialization or at runtime, but also to retrieve option’s values. Please refer to section 3 for details about these options.

The server must be configured properly before running it. Typically following options must be initialized using wip_netInitOpts or wip_netSetOpts functions:

- WIP_NET_OPT_DHCPS_ADDR
- WIP_NET_OPT_DHCPS_NB_ADDR
- WIP_NET_OPT_DHCPS_FIRST_ADDR
- WIP_NET_OPT_DHCPS_SUBNET_MASK

Note that the following options can only be set when the server is stopped:

- WIP_NET_OPT_DHCPS_NB_ADRR
- WIP_NET_OPT_DHCPS_FIRST_ADRR
- WIP_NET_OPT_DHCPS_SUBNET_MASK
- WIP_NET_OPT_DHCPS_GLOB_OPT

The DHCP server can then be activated with the option WIP_NET_OPT_DHCPS.

4.19.3. DHCP options supported by the server

To configure the list of options supported by the DHCP server, the following sequence of tag, size and pointer to memory should be used :

- tag – identifies the option (cf. RFC 2132)
- size – is the size of option's value
- pointer – must point to memory where is stored option's value (data must be persistent)

The sequence must be terminated with the special TAG_END tag and it may not exceed 32 options.

The following example shows what the sequence should look like.

```
static u8 ds_mask[4] = {255, 255, 255, 128};
static u8 ds_gate[4] = {192, 9, 200, 2};
static u8 ds_dnss[8] = {192,9,200,2,  193,131,248,2};
static u8 ds_dnme[15]= {“domain-name.com”};

wip_netDhcpOption_t optList[]={
  /* mask of the subnet managed by the server */
  TAG_SUBNET_MASK,     4,   ds_mask,

  /* gateway */
  TAG_GATEWAY,         4,   ds_gate,

  /* IP addresses of DNS servers */
  TAG_DOMAIN_SERVER,   8,   ds_dnss,

  /* Domain name */
  TAG_DOMAIN_NAME,     15,  dns_dnme,

  TAG_END
};
```

To allow the stack to fill in the options at runtime, the sequence must be passed with the WIP_NET_DHCPS_GLOB_OPT option when the DHCP server is configured.

The server does not copy the list of options but works directly with the given list. As a consequence all this information must be stored in a persistent way.

From the parameter request list sent by a DHCP client, the DHCP server will treat only those configured in its option list. It will omit any parameter it cannot provide.

4.19.4. DHCP server restrictions

The DHCP server is only able to manage one range of IP addresses and consequently only one subnet.

The DHCP server cannot work across routers or through the intervention of BOOTP relay agents. All the addresses of the range must belong to the managed subnet.

When allocating a new address the DHCP server does not probe the reused address before allocating the new one (e.g. with an ICMP echo request.).

All addresses given to the server must be in network-byte order.

4.20. DUAL PDP Support

The DUAL PDP context supports using simultaneous 2 GPRS bearers.

The DUAL PDP CONTEXT has the following features:

- Allowed to configure and use 2 independents PDP context with different address
- Each GPRS bearer has this own event handler.

IP Stack has only one default route. Whenever a GPRS bearer is started, default route will be automatically created on this bearer interface.

Application has to define manually a static route on other bearer.

Static route can be defined and removed with following Internet Library API:

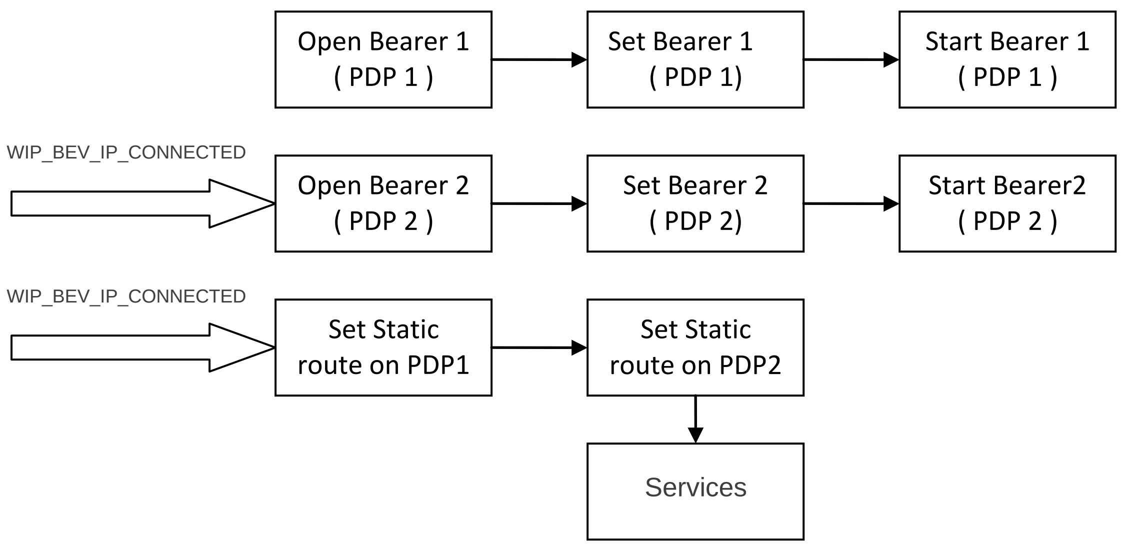
- wip_ipRouteAdd()
- wip_ipRouteDel()

4.20.1. Restriction

A GPRS bearer can be only started, stopped or closed if no other GPRS bearer is in transition progress (Connecting, Disconnecting or peer disconnecting).

4.20.2. Default use case

Multiple PDP context (GPRS Bearers) must be started (activate) one by one. The second bearer shall be started when the first one is not in CONNECTING, DISCONNECTING or PEER_DISCONNECTING states. Else WIP_BERR_BAD_STATE is returned. Normal way is to activate the second PDP after WIP_BEV_IP_CONNECTED event has been received from the first bearer.



```
// Open and set the first bearer
wip_bearerOpen ( &bgprs1 , "GPRS" , evh_bearer , NULL ) ;
wip_bearerSetOpts ( bgprs1, WIP_BOPT_GPRS_APN, GPRS_APN,
                        WIP_BOPT_LOGIN,       GPRS_USER,
                        WIP_BOPT_PASSWORD,    GPRS_PASSWORD,
                        WIP_BOPT_END ) ;

// Start the first GPRS bearer
wip_bearerStart(bgprs1);

// First GPRS event handler
static void evh_bearer ( wip_bearer_t b, s8 event, void *ctx )
{
    if( event == WIP_BEV_IP_CONNECTED )
    {
        // Open, set and start the second GPRS bearer
        wip_bearerOpen ( &bgprs2 , "GPRS2", evh_bearer2 , NULL );
        wip_bearerSetOpts ( bgprs2, WIP_BOPT_GPRS_APN, GPRS_APN2,
                            WIP_BOPT_LOGIN,       GPRS_USER2,
                            WIP_BOPT_PASSWORD,    GPRS_PASSWORD2,
                            WIP_BOPT_END ) ;

        wip_bearerStart( bgprs2 ) ;
    }
}

// Second GPRS event handler
static void evh_bearer2 ( wip_bearer_t b, s8 event, void *ctx )
{
    if( event == WIP_BEV_IP_CONNECTED )
    {
    }
}
```

The default route is the last PDP context activated. Static routes shall be defined manually.

```
// To filter only 1 IP address
#define ROUTE_MASK      0xffffffff

// Get IP address of each Bearer
wip_bearerGetOpts  (  bgprs1  ,  WIP_BOPT_IP_ADDR  ,  &IP_bgprs1  ,
WIP_BOPT_END ) ;
wip_bearerGetOpts  (  bgprs2  ,  WIP_BOPT_IP_ADDR  ,  &IP_bgprs2  ,
WIP_BOPT_END ) ;

// configure ROUTE for DST_SERVER on bearer 1 (IP_bgprs1)
wip_inet_aton ( DST_SERVER_1_IP , &ip1 );
wip_ipRouteAdd ( ip1 , ROUTE_MASK , IP_bgprs1 );

// configure ROUTE for DST_SERVER on bearer 2 (IP_bgprs2)
wip_inet_aton ( DST_SERVER_2_IP , &ip2 );
wip_ipRouteAdd ( ip2 , ROUTE_MASK , IP_bgprs2 );
```

4.21. DNS Proxy

The DNS proxy serves as a default DNS server for DHCP clients on the LAN network when the modem is used as a GPRS gateway. It relies on an upstream DNS server at an ISP to perform the DNS lookups.

A DNS proxy is needed because the address of the DNS servers is generally not known at time clients get their DHCP lease, and there is no way for the DHCP server to update the configuration of the clients when this information is obtained.

To deal with these characteristics, the DHCP server on the gateway can be configured to give the address of the DNS proxy (local address of the gateway) as the DNS server to DHCP clients. The DNS proxy makes name resolution transparent from the point of view of devices on the LAN network.

Once the internet connection is established, the DNS proxy will relay DNS requests to the address of the Internet DNS server obtained for the connection, and replies as a DNS resolver to the client device on the network.

The DNS proxy has the following features:

- Forwarding of UDP based DNS requests and replies from a local network to an external DNS server.
- Primary and secondary DNS servers.
- Support of UDP requests larger than 512 bytes.
- Optional support of TCP based DNS requests.
- Configurable timeouts.
- Follows RFC 5625 guidelines.

The DNS proxy works on top of Internet Library UDP/TCP channels.

4.21.1. Required Header File

The header file for the DNS proxy interface definitions is wip_dnsproxy.h

4.21.2. The wip_dnsProxyCreateOpts function

The wip_dnsProxyCreateOpts function creates and initializes a DNS proxy with application defined options. The proxy is activated by this function but it can forward messages only when a valid DNS server address is configured.

The Internet Library library must have been initialized by wip_netInit() or wip_netInitOpts() before calling this function.

Prototype

```
int wip_dnsProxyCreateOpts( wip_dnsProxy_t * hnd_p,
                            int              optid1,
                                             ... );
```

Parameters

`hnd_p :`

Out: A handle to the created DNS proxy on success.

`optid1:`

In: First option in the list of options.

`... :`

In: List of option names followed by option values. The list must be terminated by `WIP_DNSPROXY_OPT_END` even if no option is provided.

The following options are currently defined:

Option	Value	Description
WIP_DNSPROXY_OPT_END	None	End of option list
WIP_DNSPROXY_OPT_ADDR	wip_in_addr_t	Listening address of proxy. Default : 0.0.0.0 (any address)
WIP_DNSPROXY_OPT_PORT	u16	Listening port number of proxy. Default : 53
WIP_DNSPROXY_OPT_EXPIRE	u32	Expiration timeout of requests. The minimum value is one second. Default : 5 seconds
WIP_DNSPROXY_OPT_MAXRMIT	u32	Maximum number of retransmissions before switching to secondary server. The minimum value is one retransmission. Default : 3
WIP_DNSPROXY_OPT_MAXREQS	u32	Maximum number of simultaneous requests. The minimum value is one request. Default : 8
WIP_DNSPROXY_OPT_MAXSIZE	u32	Maximum size of a DNS over UDP message. The minimum value is 512 bytes. Default : 1024
WIP_DNSPROXY_OPT_DNS1	wip_in_addr_t	Address of primary DNS server. Default : 0.0.0.0 (none)
WIP_DNSPROXY_OPT_DNS2	wip_in_addr_t	Address of secondary DNS server. Default : 0.0.0.0 (none)
WIP_DNSPROXY_OPT_TCP	bool	Enable TCP requests forwarding. *Note: This option is reserved for future use.* Default : FALSE

Returned Values

The function returns 0 on success. In case of an error, a negative error code is returned as described below:

Error Code	Description
WIP_DNSPROXY_ERR_OPTION	Invalid option
WIP_DNSPROXY_ERR_PARAM	Invalid option value
WIP_DNSPROXY_ERR_MEMORY	Memory allocation error

Error Code	Description
WIP_DNSPROXY_ERR_CHANNEL	UDP channel creation failure
WIP_DNSPROXY_ERR_TCP_CHANNEL	TCP channel creation failure
WIP_DNSPROXY_ERR_TIMER	ADL timer creation failure

4.21.3. The wip_dnsProxyClose function

The wip_dnsProxyClose function terminates the DNS proxy; all channels are closed and all allocated memory and resources are released.

The DNS proxy must be closed before terminating the Internet Library library with wip_netExit().

Prototype

```
int wip_dnsProxyClose( wip_dnsProxy_t hnd);
```

Parameters

hnd:

In: The DNS proxy handle.

Returned Values

The function returns 0 on success. In case of an error, a negative error code is returned as described below:

Error Code	Description
WIP_DNSPROXY_ERR_PARAM	Invalid option value

4.21.4. The wip_dnsProxySetOpts function

The wip_dnsProxySetOpts function is used to set or change configuration options of the DNS proxy. Not all options can be changed after the proxy has been initialized.

All pending requests are flushed when the configuration of the DNS proxy is changed. The current DNS server is reset to the primary server.

Prototype

```
int wip_dnsProxySetOpts( wip_dnsProxy_t hnd,
                         int            optid1,
                                         ... );
```

Parameters

hnd:

In: The DNS proxy handle.

optid1:

In: First option in the list of options.

... :

In: List of option names followed by option values. The list must be terminated by `WIP_DNSPROXY_OPT_END` even if no option is provided.

The supported options and their default values are defined in the table below. But only the following options can be changed after the initialization of the proxy:

- `WIP_DNSPROXY_OPT_EXPIRE`, expiration timeout of requests.
- `WIP_DNSPROXY_OPT_MAXRMIT`, maximum number of retransmissions before switching to secondary server.
- `WIP_DNSPROXY_OPT_DNS1`, address of primary DNS server.
- `WIP_DNSPROXY_OPT_DNS2`, address of secondary DNS server.

Option	Value	Description
WIP_DNSPROXY_OPT_END	None	End of option list
WIP_DNSPROXY_OPT_ADDR	wip_in_addr_t	Listening address of proxy. Default : 0.0.0.0 (any address)
WIP_DNSPROXY_OPT_PORT	u16	Listening port number of proxy. Default : 53
WIP_DNSPROXY_OPT_EXPIRE	u32	Expiration timeout of requests. The minimum value is one second. Default : 5 seconds
WIP_DNSPROXY_OPT_MAXRMIT	u32	Maximum number of retransmissions before switching to secondary server. The minimum value is one retransmission. Default : 3
WIP_DNSPROXY_OPT_MAXREQS	u32	Maximum number of simultaneous requests. The minimum value is one request. Default : 8
WIP_DNSPROXY_OPT_MAXSIZE	u32	Maximum size of a DNS over UDP message. The minimum value is 512 bytes. Default : 1024
WIP_DNSPROXY_OPT_DNS1	wip_in_addr_t	Address of primary DNS server. Default : 0.0.0.0 (none)
WIP_DNSPROXY_OPT_DNS2	wip_in_addr_t	Address of secondary DNS server. Default : 0.0.0.0 (none)
WIP_DNSPROXY_OPT_TCP	bool	Enable TCP requests forwarding. Default : FALSE

Returned Values

The function returns 0 on success. In case of an error, a negative error code is returned as described below:

Error Code	Description
WIP_DNSPROXY_ERR_OPTION	Invalid option
WIP_DNSPROXY_ERR_PARAM	Invalid option value

4.21.5. The wip_dnsProxyGetOpts function

The wip_dnsProxyGetOpts function retrieves configuration options of the DNS proxy.

Prototype

```
int wip_dnsProxyGetOpts( wip_dnsProxy_t hnd,
                         int            optid1,
                                        ... );
```

Parameters

hnd:

In: The DNS proxy handle.

optid1:

In: First option in the list of options

... :

In: List of option names followed by option values. The list must be terminated by `WIP_DNSPROXY_OPT_END` even if no option is provided.

The following options are currently defined:

Option	Value	Description
WIP_DNSPROXY_OPT_END	None	End of option list
WIP_DNSPROXY_OPT_ADDR	wip_in_addr_t	Listening address of proxy. Default : 0.0.0.0 (any address)
WIP_DNSPROXY_OPT_PORT	u16	Listening port number of proxy. Default : 53
WIP_DNSPROXY_OPT_EXPIRE	u32	Expiration timeout of requests. The minimum value is one second. Default : 5 seconds
WIP_DNSPROXY_OPT_MAXRMIT	u32	Maximum number of retransmissions before switching to secondary server. The minimum value is one retransmission. Default : 3
WIP_DNSPROXY_OPT_MAXREQS	u32	Maximum number of simultaneous requests. The minimum value is one request. Default : 8
WIP_DNSPROXY_OPT_MAXSIZE	u32	Maximum size of a DNS over UDP message. The minimum value is 512 bytes. Default : 1024
WIP_DNSPROXY_OPT_DNS1	wip_in_addr_t	Address of primary DNS server. Default : 0.0.0.0 (none)

Option	Value	Description
WIP_DNSPROXY_OPT_DNS2	wip_in_addr_t	Address of secondary DNS server. Default : 0.0.0.0 (none)
WIP_DNSPROXY_OPT_TCP	bool	Enable TCP requests forwarding. Default : FALSE

Returned Values

The function returns 0 on success. In case of an error, a negative error code is returned as described below:

Error Code	Description
WIP_DNSPROXY_ERR_OPTION	Invalid option

4.22. Ethernet Bearer Management

This section describes the extensions of the bearer interface needed to support Ethernet network interfaces.

4.22.1. Required Header File

The header file for the Ethernet bearer management is wip_eth.h.

4.22.2. The wip_ethAddr_t Type

The wip_ethAddr_t type stores a 6-byte MAC address:

```
typedef u8 wip_ethAddr_t[6];
```

4.22.3. The wip_ethLink_e Type

The wip_ethLink_e type defines the status or configuration of the Ethernet link:

```
typedef enum {
  WIP_ETH_LINK_AUTONEG         = 0,          /* Select auto negotiation */
  WIP_ETH_LINK_UNKNOWN         = 0,          /* Status not available */
  WIP_ETH_LINK_DOWN            = 1,          /* Link down */
  WIP_ETH_LINK_10BASE2         = 2,          /* 10BASE-2 */
  WIP_ETH_LINK_10BASE5         = 3,          /* 10BASE-5 (AUI)*/
  WIP_ETH_LINK_10BASET         = 4,          /* 10BASE-T half duplex */
  WIP_ETH_LINK_10BASET_FD      = 5,          /* 10BASE-T full duplex */
  WIP_ETH_LINK_100BASETX       = 6,          /* 100BASE-TX half duplex */
  WIP_ETH_LINK_100BASETX_FD    = 7           /* 100BASE-TX full duplex */
} wip_ethLink_e;
```

4.22.4. The wip_ethLinkCap_e Type

The wip_ethLinkCap_e type stores the capabilities of the Ethernet link advertised during auto-negotiation:

```
typedef enum {
  WIP_ETH_LINKCAP_10BASET        = 0x0001,  /* 10BASE-T half duplex */
  WIP_ETH_LINKCAP_10BASET        = 0x0002,  /* 10BASE-T full duplex */
  WIP_ETH_LINKCAP_100BASETX      = 0x0004,  /* 100BASE-TX half duplex */
  WIP_ETH_LINKCAP_100BASETX_FD   = 0x0008,  /* 100BASE-TX full duplex */
} wip_ethLinkCap_e;
```

4.22.5. Ethernet Bearer Options

The Ethernet bearer is an extension of the existing IP bearer management. The Ethernet bearer specific options can be set with wip_bearerSetOpts () and wip_bearerGetOpts () APIs. The following options are supported by the Ethernet bearers.

Option	Value	Description
WIP_BOPT_ETH_ADDR	wip_ethAddr_t *	MAC address of the interface, the address cannot be changed after bearer is started
WIP_BOPT_ETH_LINK_CFG	wip_ethLink_e	Configuration of the link, the default is auto-negotiation.
WIP_BOPT_ETH_LINK	wip_ethLink_e	Current link status (get only).
WIP_BOPT_ETH_LINKADV	wip_ethLinkCap_e	A mask of link capabilities to advertise during link auto-negotiation.
WIP_BOPT_ETH_PROMISC	bool	Enable promiscuous mode (reception of all packets).
WIP_BOPT_ETH_ALLMULTI	bool	Enable reception of all multicast packets.

In addition, Ethernet bearer also supports the standard options defined below. The bearer supports static configuration using WIP_BOPT_IP_ADDR, NETMASK, GW, DNS1 and DNS2 options, and also auto-configuration using WIP_BOPT_IP_DHCP, SETDNS and SETGW. When auto-configuration is enabled, a DHCP server must be present on the network.

Option	Value	Description
WIP_BOPT_NAME	ascii *	Name of bearer device (get only). default:0
WIP_BOPT_TYPE	wip_bearerType_e	Type of bearer (get only): WIP_BEARER_ETHER
WIP_BOPT_IFINDEX	wip_ifindex_t	Index of network interface (get only).
WIP_BOPT_ERROR	s8	Error code indicating the cause of the disconnection (get only). default:0
WIP_BOPT_END	none	End of option list.
IP Options		
WIP_BOPT_IP_ADDR	wip_in_addr_t	Local IP address, suggested IP address when DHCP is enabled. default:0

Option	Value	Description
WIP_BOPT_IP_NETMASK	wip_in_addr_t	Network mask.
WIP_BOPT_IP_GW	wip_in_addr_t	Address of default gateway.
WIP_BOPT_IP_DNS1	wip_in_addr_t	Address of primary DNS server. default:0
WIP_BOPT_IP_DNS2	wip_in_addr_t	Address of secondary DNS server. default:0
WIP_BOPT_IP_DHCP bool Enable auto-configuration of IP address and netmask with DHCP.	bool	Enable auto-configuration of IP address and netmask with DHCP. default:TRUE
WIP_BOPT_IP_SETDNS	bool	Auto-configure DNS resolver using DHCP information, WIP_BOPT_IP_DHCP must also be enabled. default:TRUE
WIP_BOPT_IP_SETGW	bool	Set default gateway using DHCP information, WIP_BOPT_IP_DHCP must also be enabled. default:TRUE

4.23. Driver Specific Options

The functions wip_bearerSetDrvOption() and wip_bearerGetDrvOption() allow an application to set get and get driver-specific options. The options are not interpreted by Internet Library and are passed unmodified to the driver, the behavior of the options must be defined by the driver.

4.23.1. The wip_bearerGetDrvOption function

This function is used to get driver specific options. each driver can define its own set of functions.

Note: This function can be called when the bearer is not started.

Prototype

```
s8 wip_bearerGetDrvOption( wip_bearer_t br, s32 optname,
                           void *optval, s32 *optlen,
                           s32 *ret);
```

Parameters

br:

In: bearer handle

optname:

In: name of option, specific to the driver.

optval:

Out: buffer filled with option value.

optlen:

In: length of buffer.

Out: length of returned option value.

ret:

Out: result code

Returned Values

The function returns 0 on success, otherwise a negative error code. On success the application must also check the result code.

4.23.2. The wip_bearerSetDrvOption function

This function is used to set driver specific options. each driver can define its own set of functions.

Note: This function can be called when the driver is not active.

Prototype

```
s8 wip_bearerSetDrvOption( wip_bearer_t br, s32 optname,
                           const void *optval, s32 optlen,
                           s32 *ret);
```

Parameters

br:

In: bearer handle

optname:

In: name of option, specific to the driver.

optval:

In: option value.

optlen:

In: length of option value.

ret:

Out: result code

Returned Values

The function returns 0 on success, otherwise a negative error code. On success the application must also check the result code.

4.24. Asynchronous Event Notification

The bearer event WIP_BEV_DRIVER is used by the driver to notify the application of a driver specific event. The meaning of the event is driver-dependant and this event is not interpreted by Internet Library.

4.25. Network Interface Driver

This section describes the network interface driver interface that allows an application to add a custom bearer. Currently only Ethernet devices are supported.

4.25.1. Required Header File

The header file for the interface driver is wip_drv.h.

4.25.2. The wip_drvData_t Type

The wip_drvData_t type stores data common to all types of drivers:

```
typedef struct {
  wip_bearerType_e  drv_type;        /* type of associated bearer */
  bool              drv_up;          /* true if bearer is started
  void             *drv_data;        /* driver specific data */
  void             *drv_priv;        /*  pointer  to  Internet  Library
                                     internal data
} wip_drvData_t;
```

4.25.3. The wip_drvEthData_t Type

The wip_drvEthData_t type stores data specific to Ethernet drivers:

```
typedef struct {
  wip_drvData_t     eth_drvdata;     /* generic driver data */
  wip_ethAddr_t     eth_addr;        /* MAC address */
  bool              eth_promisc;     /* promiscuous mode */
  bool              eth_allmulti;    /* receive all multicasts */
  int               eth_mcast_nb;    /* nb of addresses in list */
  wip_ethAddr_t     eth_mcast[];     /* list of multicast addresses */
  wip_ethLink_e     eth_link;        /* current link status */
  wip_ethLink_e     eth_linkcfg;     /* link configuration */
  wip_ethLinkCap_e  eth_linkadv;     /* link advertisement */
} wip_drvEthData_t;
```

4.25.4. The wip_drvCtl_e Type

The wip_drvCtl_e type encodes a command for the driver:

```
typedef  enum {
  /* generic commands */
  WIP_DRVCTL_UP            = 1,      /* start driver */
```

```
  WIP_DRVCTL_DOWN,                  /* stop driver */
  WIP_DRVCTL_OUTPUT,                /* packets waiting for tx */
  WIP_DRVCTL_TIMER,                 /* 2Hz timer */
  /* Ethernet specific commands */
  WIP_DRVCTL_ETH_SETFILTER = 20,    /* update input filter */
  WIP_DRVCTL_ETH_SETPHY             /* update PHY configuration */
} wip_drvCtl_e;
```

4.25.5. The wip_drvCtlHdlr_f Type

The wip_drvCtlHdlr_f type is the prototype of the driver control function. This function handles commands from the bearer manager.

-

```
typedef s32 ( *wip_drvCtlHdlr_f ) ( wip_drvData_t   *drvp,
                                    wip_drvCtl_e   cmd,
                                    void   *arg );
```

drvp:

In: Pointer to driver data, this pointer can be casted to the driver specific structure associated to the type of bearer. For example, if the bearer has the Ethernet type this pointer can be casted to a wip_drvEthData_t structure.

cmd:

In: Command name, see below for the list of supported commands. The driver should implement the following commands:

Command	Description
WIP_DRVCTL_UP	Allow the driver to send and receive data, returns OK on success, a non-zero value if initialization has failed, this command is sent when the bearer is started
WIP_DRVCTL_DOWN	Stop communication, this command is sent when the bearer is stopped
WIP_DRVCTL_OUTPUT	Indicate to the driver that a new buffer has been queued from transmission; driver can call wip_drvBufDequeue to get the next buffer to transmit.
WIP_DRVCTL_TIMER	Called WIP_DRV_TIMERHZ (2) times per second when the driver is enabled. Ethernet drivers can update eth_link status during that call.
WIP_DRVCTL_SETOPTION	Set driver specific options, the command argument is a pointer to a wip_drvCtlOption_t structure which contains the option name, a pointer to the option value and the lenght of option value. The driver should return 0 if the option has been successfuly set, otherwise a non-zero value should be returned. This command can be called when the driver is stopped.
WIP_DRVCTL_GETOPTION	Get driver specific options, the command argument is a pointer to a wip_drvCtlOption_t structure which contains the option name and the maximum lenght of option value. The driver must call wip_drvOptionCpy() to copie the value of the option to the caller memory and return 0 on success, otherwise a non-zero value should be returned. This command can be called when the driver is stopped.
WIP_DRVCTL_UNSUBSCRIBE	Called when the driver is unsubscribed. Any driver specific data allocated during subscription should be released.

The following error codes may be returned by the WIP_DEVCTL_UP command:

Error Code	Description
WIP_BERR_DEV	Error during initialization of driver

The Ethernet drivers should implement following additional commands.

Command	Description
WIP_DRVCTL_ETH_SETFILTER	Indicates that Ethernet input filter has changed: promiscuous and all multicasts flags, multicast filter (eth_promisc, eth_allmulti, eth_mcast_nb, eth_mcast fields of driver data)
WIP_DRVCTL_ETH_SETPHY	Ethernet PHY configuration has changed (eth_linkcfg, eth_linkadv fields of driver data)

`arg:`

In: Command argument (not used)

The MAC address is stored in the eth_addr field. There are two way of setting the address:

- The application can set the address using the bearer option WIP_BOPT_ETH_ADDR, before the bearer is started.
- The driver can set the address during the WIP_DRVCTL_UP command, for example if the address is read from an eeprom connected to the Ethernet controller.

In both cases the address cannot be changed after the driver is started.

The fields of the wip_drvEthData_t structure are read-only except for eth_link which should reflect the current status of the link and may be updated during WIP_DRVCTL_TIMER command. The eth_addr may be updated during WIP_DRVCTL_UP command.

4.26. The wip_drvSubscribe Function

The wip_drvSubscribe function declares a network interface driver to the bearer manager. When a bearer is created, it allows the application to configure and manage the network interface.

4.26.1. Prototype

```
s32 wip_drvSubscribe ( const char    *brname,
                       wip_bearerType_e   type,
                       wip_drvCtlHdlr_f   drvctl,
                       void   *data );
```

4.26.2. Parameters

brname:

In: Name of bearer to create

type:

In: Type of bearer

drvctl:

In: Driver control function

data:

In: Driver specific data, copied into drv_data field of the wip_drvData_t structure.

4.26.3. Returned Values

The function returns a positive or null handle on success, otherwise a negative error code.

Error Code	Description
ADL_RET_ERR_PARAM	A parameter is not valid: the type of bearer is not supported, the bearer name already exists, the bearer name is too long, no control function is provided
ADL_RET_ERR_NO_MORE_HANDLES	There is no available handle for a new driver

Note: *In the current implementation only one driver can be subscribed, the type of bearer must be WIP_BEARER_ETHER (Ethernet).*

The commands/events will be received in the driver control function only when the bearer is started

4.27. The wip_drvUnsubscribe Function

The wip_drvUnsubscribe function removes a network interface driver and its associated bearer. The bearer must be closed before it can be removed.

4.27.1. Prototype

```
s32 wip_drvUnsubscribe ( s32   drvHandle );
```

4.27.2. Parameters

drvHandle:

In: Driver handle

4.27.3. Returned Values

The function returns zero on success, otherwise a negative error code.

Error Code	Description
ADL_RET_BAD_HDL	The handle is not a valid driver handle
ADL_RET_ERR_BAD_STATE	The associated bearer is not closed

4.28. The wip_drvOptionCpy function

The wip_drvOptionCpy() function is used by the driver to write the value of an option when the WIP_DRVCTL_GETOPTION command is called. This allows the function to return values to a calling task in protected memory.

This function must be called only within the control function during processing of the WIP_DRVCTL_GETOPTION command.

Prototype

```
s32 wip_drvOptionCpy( wip_drvData_t *drvp, const void *optval,
                      int optlen);
```

Parameters

drvp:

In: Pointer to driver data, this pointer can be casted to the driver specific structure associated to the type of bearer. For example, if the bearer has the Ethernet type this pointer can be casted to a wip_drvEthData_t structure.

optval:

In: option value.

optlen:

In: length of option value.

Returned Values

None

4.29. Buffer Management Functions

The following functions are used by the driver to handle network buffers. Each buffer contains a single packet. The format of the packet is dependant to the type of device.

The driver can only access the buffers for reading, writing to the buffers can be achieved by using adl bus write functions or by using wip_drvBufMemCpy() and wip_drvBufSetLen().

4.29.1. The wip_drvBuf_t Structure

The wip_drvBuf_t structure stores a network buffer. The structure exports the following fields to the driver, other fields are private. The structure is read-only; the driver can change the value of buf_datalen by calling wip_drvBufSetLen():

```
typedef struct {
/* ... */                          /* private fields */
  void *buf_pad1;
  Int   buf_pad2;
  u8   *buf_datap;                 /* pointer to buffer data */
  int   buf_datalen;               /* length of data */
} wip_drvBuf_t;
```

When a buffer is allocated by wip_drvBufAlloc() the field buf_datalen is initialized with the requested length.

In the case of Ethernet devices, a network buffer contains an Ethernet packet. The Ethernet packet contains 14-bytes of Ethernet header followed by 0-1500 bytes of data. The Ethernet CRC must not be included.

4.29.2. The wip_drvBufAlloc Function

The wip_drvBufAlloc function allocates a network buffer. The buf_datap and buf_datalen fields are initialized according to the given length. The driver can then change the length of the allocated buffer by calling wip_drvBufSetLen(), but only with a smaller value.

Prototype

```
wip_drvBuf_t *wip_drvBufAlloc ( wip_drvData_t   *drvp,
                                int   len );
```

Parameters

drvp:

In: Pointer to driver data

len:

In: Maximum length of buffer data

Returned Values

The function returns

- a pointer to the allocated buffer on success
- a NULL pointeron error

Allocation fails if there is no more free buffer or if the requested length is out of range.

4.29.3. The wip_drvBufFree Function

The wip_drvBufFree function releases a network buffer.

Prototype

```
void wip_drvBufFree ( wip_drvData_t   *drvp,
                      wip_drvBuf_t   *bufp );
```

Parameters

drvp:

In: Pointer to driver data

bufp:

In: Pointer to buffer

Returned Values

None

4.29.4. The wip_drvBufDequeue Function

The wip_drvBufDequeue() function dequeues the next buffer to be transmisted by the driver.

Prototype

```
wip_drvBuf_t *wip_drvBufDequeue ( wip_drvData_t   *drvp );
```

Parameters

`drvp:`

In: Pointer to driver data

Returned Values

The function returns

- a pointer to the next buffer to transmit on success
- a NULL pointer if the driver output queue is empty

4.29.5. The wip_drvBufEnqueue Function

The wip_drvBufEnqueue function enqueues a received buffer for input processing by the TCP/IP stack. After this call the driver must not access the buffer.

Prototype

```
void wip_drvBufEnqueue ( wip_drvData_t   *drvp,

                         wip_drvBuf_t   *bufp );
```

Parameters

drvp:

In: Pointer to driver data

bufp:

In: Pointer to buffer

Returned Values

None

4.29.6. The wip_drvBufMemCpy function

The wip_drvBufMemCpy() function is used to copy bytes from the driver memory to the protected memory of a network buffer. The driver is not allowed to directly write into a network buffer.

Prototype

```
void wip_drvBufMemCpy( wip_drvData_t *drvp, wip_drvBuf_t *bufp,
                       int offset, int len, const void *src);
```

Parameters

drvp:

In: pointer to driver data

bufp:

In: pointer to buffer

offset:

In: offset of data to copy in destination buffer

len:

In: length of data

src:

In: pointer to source data

Returned Values

None

4.29.7. The wip_drvBufSetLen function

The wip_drvBufSetLen() function is used to change the length of a network buffer stored in buf_datalen field. The driver is not allowed to directly to directly write to the buffer structure.

Prototype

```
void wip_drvBufSetLen( wip_drvData_t *drvp, wip_drvBuf_t *bufp,
                       int len);
```

Parameters

drvp:

In: pointer to driver data

bufp:

In: pointer to buffer

len:

In: new length of buffer data

Returned Values

None.

4.30. Asynchronous Event Notification

The following functions can be used by a driver to report asynchronous events to the application. The meaning of the event is driver-specific.

The application is notified by receiving a bearer event WIP_BEV_DRIVER.

4.30.1. The wip_drvSetEvent function

The wip_drvSetEvent() function generates a WIP_BEV_DRIVER event to the associated bearer. Multiple calls to wip_drvSetEvent will not necessary generate the same number of bearer events; it is only guaranted that at least one notification is received by the application for the last wip_drvSetEvent call.

Prototype

```
void wip_drvSetEvent( wip_drvData_t *drvp);
```

Parameters

drvp:

In: pointer to driver data

Returned values

None.

4.31. Interrupt Handling Functions

The driver must use the following functions if it needs to use an interrupt handler:

- wip_drvIsrSubscribe()
- wip_drvIsrUnsubscribe()
- wip_drvIrqDisable()
- wip_drvIrqRestore()

4.31.1. The wip_drvIrqMode_e Type

The wip_drvIrqMode_e type defines the trigger mode of the interrupt:

```
typedef enum {
  WIP_DRV_IRQ_TRIGGER_HIGH_LEVEL,       /* level triggered, active high */
  WIP_DRV_IRQ_TRIGGER_LOW_LEVEL,        /* level triggered, active low */
  WIP_DRV_IRQ_TRIGGER_RISING_EDGE,      /* triggered on rising edge */
  WIP_DRV_IRQ_TRIGGER_FALLING_EDGE,     /* triggered on falling edge */
  WIP_DRV_IRQ_TRIGGER_ANY_EDGE          /* triggered on both edges */
} wip_drvIrqMode_e;
```

Note: *The current implementation only supports edge triggered interrupts WIP_DRV_IRQ_TRIGGER_RISING_EDGE, WIP_DRV_IRQ_FALLING_EDGE and WIP_DRV_IRQ_TRIGGER_ANY_EDGE.*

4.31.2. The wip_drvIsrHdlr_f Type

The interrupt handler has the following prototype:

```
typedef ( void *wip_drvIsrHdlr_f ) ( wip_drvData_t   *drvp );
```

drvp:

In: is the pointer to the driver data.

4.31.3. The wip_drvIsrSubscribe Function

The wip_drvIsrSubscribe function attaches an interrupt handler to an external interrupt. This call also enables the external interrupt.

Prototype

```
s32 wip_drvIsrSubscribe ( wip_drvData_t   *drvp,
                          wip_drvIsrHdlr_f   isrHandler,
                          u8   irq,
                          wip_drvIrqMode_e   mode );
```

Parameters

drvp:

In: Pointer to driver data

isrHandler:

In: Interrupt handler function

irq:

In: Interrupt identifier

mode:

In: Interrupt trigger mode

Returned Values

The function returns

- a positive or null handle on success
- In case of an error, a negative error code as described below:

Error Code	Description
ADL_RET_ERR_PARAM	The interrupt identifier is not valid, the trigger mode is not valid
ADL_RET_ERR_NO_MORE_HANDLE	There is no available handle
ADL_RET_ERR_NOT_SUBSCRIBED	The interrupt context call stack size was not supplied by the application
ADL_RET_ERR_BAD_STATE	The function is called in RTE mode
ADL_RET_ERR_NOT_SUPPORTED	The Real Time enhancement feature is not enabled on the embedded module (deprecated error code, kept for compatibility with firmware prior to 7.45)

4.31.4. The wip_drvIsrUnsubscribe Function

The wip_drvIsrUnsubscribe function disables the external interrupt associated with the given handle and removes the attached interrupt handler.

Prototype

```
s32 wip_drvIsrUnsubscribe ( s32   isrHandle );
```

Parameters

isrHandle:

In: Interrupt handle

Returned Values

The function returns

- zero on success
- a negative error code

4.31.5. The wip_drvIrqDisable Function

The wip_drvIrqDisable function disables external interrupts.

Prototype

```
s32 wip_drvIrqDisable ( void );
```

Parameters

None

Returned Values

The function returns an integer that must be passed to wip_drvIntrRestore() in order to restore the interrupt state.

4.31.6. The wip_drvIrqRestore Function

The wip_drvIrqRestore function restores the interrupt status after a call to wip_drvIrqDisable function.

Prototype

```
void wip_drvIrqRestore ( s32   oldstate );
```

Parameters

oldtstate:

In: The value previously returned by wip_drvIrqDisable function

Returned Values

None

4.32. Ethernet Bearer Temporal Diagram

The following diagrams describe all the behaviours supported by the interface.

4.32.1. Driver Activation

To open, configure and start the related bearer first the driver has to be registered with the wip_drvSubscribe () API.

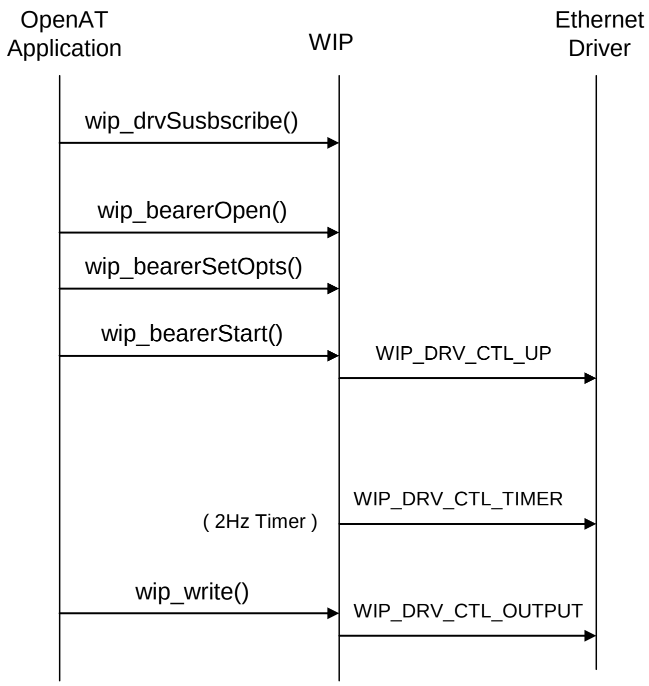


Figure 7. Ethernet Driver Activation Temporal Diagram

4.32.2. Driver Shutdown

The driver will shut down if the bearer is stopped. Additionaly the application can unsubscribe the driver to release all resources.

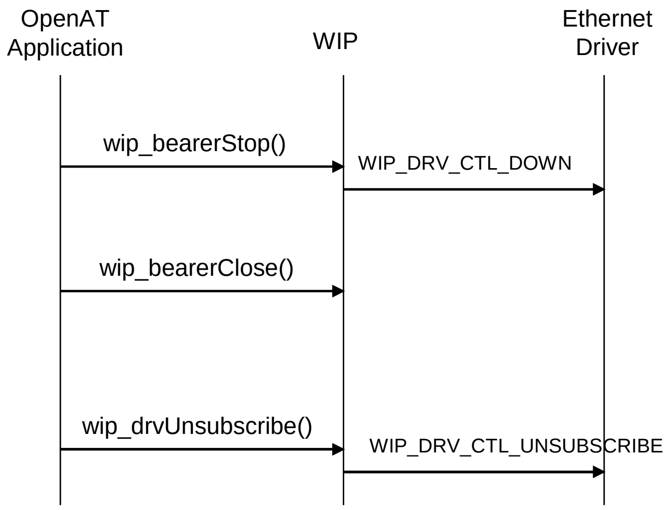


Figure 8. Ethernet Driver Shutdown Temporal Diagram

4.32.3. Driver Setting and Retrieval (Set and Get)

The application uses wip_bearerSetDrvOption and wip_bearerGetDrvOption to set and get driver specific options.

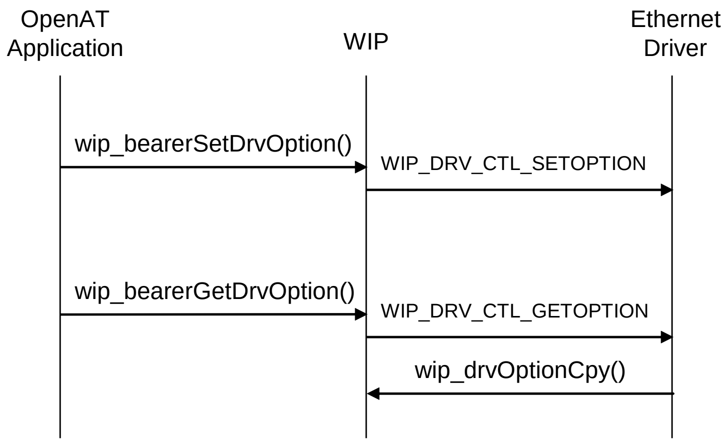


Figure 9. Ethernet Driver Set and Get Temporal Diagram

5. Internet Protocol Support Library

The Internet Protocol support library provides support for internet addresses.

5.1. Required Header File

The header file for the IP Support Library related functions is wip_inet.h.

5.2. The wip_in_addr_t Structure

The wip_in_addr_t type stores a 32-bit IPv4 address in network-byte order.

```
typedef u32 wip_in_addr_t;
```

5.3. The wip_inet_aton Function

The wip_inet_aton function converts an internet address in standard dot notation to a wip_in_addr_t type.

5.3.1. Prototype

```
bool wip_inet_aton ( const ascii   *str,

                     wip_in_addr_t   *addr );
```

5.3.2. Parameters

str:

In: Null terminated string that contains the IP address to convert in dot notation

addr:

Out: Filled with converted IP address

5.3.3. Returned Values

The function returns

- TRUE if the provided string contains a valid IP address
- FALSE if it does not contain a valid IP address

5.4. The wip_inet_ntoa Function

The wip_inet_ntoa function converts an internet address to a string in the standard dot notation.

5.4.1. Prototype

```
bool wip_inet_ntoa ( wip_in_addr_t   addr,
                     ascii   *buf,
                     u16   buflen );
```

5.4.2. Parameters

addr:

In: IP address

buf:

In: Pointer to destination buffer

buflen:

In: Length of destination buffer

5.4.3. Returned Values

The function returns

- TRUE if the provided buffer is large enough to store the result string
- else FALSE is returned

6. Socket Layer

6.1. Common Types

6.1.1. Channels

Channels are opaque to the user and must be manipulated only through API functions.

```
typedef struct channel *wip_channel_t;
```

6.1.2. Event Structure

A channel event is composed of a constant indicating the kind of event which happened, as described by the kind field. Every kind of event corresponds to a specific set of data. These specific data types are gathered in specific structures, which in turn are included in the channelEvent structure through a union content. If event.kind is WIP_CEV_READ, only the event.content.read union field is relevant. If kind is WIP_CEV_WRITE, event.content.write is relevant; WIP_CEV_PEER_CLOSE corresponds to event.content.peer_close, WIP_CEV_ERROR to event.content.error, and WIP_CEV_PING to event.content.ping.

```
typedef struct wip_event_t {
  enum wip_event_kind_t {
    WIP_CEV_DONE,
    WIP_CEV_ERROR,
    WIP_CEV_OPEN,
    WIP_CEV_PEER_CLOSE,
    WIP_CEV_PING,
    WIP_CEV_READ,
    WIP_CEV_WRITE,                /*File-handling related events*/
    WIP_CEV_CLOSE_DIR,
    WIP_CEV_READ_DIR,
    WIP_CEV_REWIND_DIR,
    WIP_CEV_LAST = WIP_CEV_REWIND_DIR
  } kind;
  wip_channel_t channel;
  union wip_event_content_t {
    struct wip_event_content_read_t {
      u32 readable;              /* how many bytes can be read */
    } read;
    struct wip_event_content_write_t {
      u32 writable;              /* how many bytes can be written */
    } write
    struct wip_event_content_ping_t {
```

```
      int packet_idx;             /* Index of the packet in the sent
                                     sequence*/
      u32 response_time;          /* Time taken by the echo to come back, in
                                     ms. */
      bool timeout;               /*  Did  the  echo  take  too  long  to  come
                                  back?
                                     If timeout is true, response_time is
                                     meaningless  (and set to 0) */
    } ping;

    struct wip_event_content_error_t {
      wip_error_t errnum;         /* Error */
    } error;
    struct wip_event_content_done_t {
      int result;
      int aux;
    } done
  } content;
} wip_event_t;
```

6.1.3. Opaque Channel Type

Channels are not to be inspected directly by the user, who might only interact with them through API functions. The corresponding type is therefore opaque to them.

```
typedef struct channel *wip_channel_t;
/* The [wip_channel_struct_t] structure is not declared in the public API.
The user can only work with pointers as abstract datatypes.*/
```

6.1.4. Event Handler Callback wip_eventHandler_f

```
typedef void (*wip_eventHandler_f) ( wip_event_t   *ev,
                                     void   *ctx );
```

When a channel is created, a callback function must be passed to react to channel events. This callback type is wip_eventHandler_f, and takes the following as parameters:

ev: The structure describing the event

ctx: A pointer to user data which is passed at channel creation time. This allows the user to associate connection specific data to the channel. If not required it will be set to NULL.

6.1.5. Options

The table below summarizedthe options that can be passed to channels through the “Opts” functions, together with their meaning, and the type of parameter(s) they take. For instance, WIP_COPT_PORT takes an s16 as a parameter. This means that when used in an option-setting context, WIP_COPT_PORT is to be followed by an s16 parameter, then by the next option (or WIP_COPT_END). When used in an option-getting context, it will be followed by a pointer to an integer, where the port number will be written.

Option	Description	Set Type	Get Type
WIP_COPT_END	Indicates that the last option of the list is reached	<none>	<none>
WIP_COPT_KEEPALIVE	Sends a NOOP command every n tenth of seconds, so that the server and any NAT on the way won't shut down the connection default:1	u32 n	u32* n
WIP_COPT_SND_BUFSIZE	Size of the emission buffer associated with a socket default: depends on the protocol.	u32	u32*
WIP_COPT_RCV_BUFSIZE	Size of the reception buffer associated with a socket default: depends on the protocol.	u32	u32*
WIP_COPT_SND_LOWAT	Minimum amount of available space that must be available in the emission buffer before triggering a WIP_CEV_WRITE event. default:1024	u32	u32*
WIP_COPT_RCV_LOWAT	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ event default:1	u32	u32*
WIP_COPT_RCV_TIMEOUT	For PING channels, timeout for ECHO requests. default:1000	u32	u32*
WIP_COPT_ERROR	Number of the last error experienced by that socket default:WM_EOK	none	s32*
WIP_COPT_NREAD	Number of bytes that can currently be read on that socket. default:0	none	u32*
WIP_COPT_NWRITE	Number of bytes that can currently be written on that socket. For a PING, size of the request default:20	u32	u32*
WIP_COPT_CHECKSUM	Whether the checksum control must be performed by an UDP socket. default:TRUE	bool	bool*

Option	Description	Set Type	Get Type
WIP_COPT_NODELAY	When set to TRUE, TCP packets are sent immediately, even if the buffer is not full enough When set to FALSE, the packets will be sent either, a) by combining several small packets into a bigger packet b) when the data is ready to send and the stack is idle Note: Data has to be buffered and managed by the user application. There is no provision in Internet Library APIs to wait for data block to be fully filled before sending it. default:FALSE	bool	bool*
WIP_COPT_MAXSEG	Maximum size of TCP packets	u32	u32*
WIP_COPT_TOS	Type of Service (cf. RFC 791) default:0	u8	u8*
WIP_COPT_TTL	Time-To-Live for packets default:64	u8	u8*
WIP_COPT_DONTFRAG	If set. UDP datagrams are not allowed to be fragmented when going through the network. default:FALSE	bool	bool*
WIP_COPT_PEEK	When true, the message is not deleted from the buffer after reading, so that it can be read again. default:FALSE	bool	none
WIP_COPT_PORT	Port occupied by this socket. default:0	u16	u16*
WIP_COPT_STRADDR	Local address of the socket. default:0	ascii	ascii *buf, u32 buf_len
WIP_COPT_ADDR	Local address of the socket, as a 32 bits integer. default:0	wip_in_addr_t	wip_in_addr_t*
WIP_COPT_PEER_PORT	Port of the peer socket. default:0	u16	u16*
WIP_COPT_PEER_STRADDR	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection default:0	ascii	ascii *buf, u32 buf_len
WIP_COPT_PEER_ADDR	Address of the peer socket, as a 32 bits integer. default:0	wip_in_addr_t	wip_in_addr_t*
WIP_COPT_TRUNCATE	Whether an UDP read operation truncated the received data, due to a lack of buffer space. default:FALSE	bool	bool*
WIP_COPT_REPEAT	Number of PING echo requests to send. default:1	s32	s32*

Option	Description	Set Type	Get Type
WIP_COPT_INTERVAL	Time between two PING echo requests, in ms. default:1000	u32	u32*
WIP_COPT_SUPPORT_READ	Fails if the channel does not support wip_read() operations. If supported, does nothing.	none	none
WIP_COPT_SUPPORT_WRITE	Fails if the channel does not support wip_write() operations. If supported, does nothing.	none	none
WIP_COPT_BOUND	Specifies whether the socket is bound[2] to a peer socket or not. default: 1	none	boolean*

2 The option WIP_COPT_BOUND is used to check whether an UDP socket is bound to any other UDP socket or not. When the UDP socket is created without specifying the IP address of the peer, then the option WIP_COPT_BOUND will be read as FALSE. This is because there is no destination IP address to communicate with. If the UDP socket is created by specifying the peer IP address, the option WIP_COPT_BOUND will be read as TRUE. This is because the peer IP address will be resolved by the DNS and the socket is said to be bounded to the peer socket. Hence this option will be read as TRUE.

Note: *It does make sense to put zero sized buffers. For instance, if user knows that the socket will be used only for sending data and never for reading data, then read buffer size can be set to zero to save some memory.*

6.2. Common Channel Functions

This section describes common channel functions that can be used for various purposes such as to close, read or write from a channel.

6.2.1. The wip_close Function

The wip_close function closes a channel.

Note: *The actual resource release does not happen immediately. Instead, the channel is put on a "closing queue" and will be closed at a safe time. This way, the user can request to close a channel at any time – even while handling an event triggered by the channel that the user wants to close.*

wip_close () should not be used to close a non-created or a closed channel. If it is used to close a non-created or a closed channel, zero will be received as return value.

Prototype

```
int wip_close ( wip_channel_t   c );
```

Parameters

c:

In: The channel that must be closed.

Returned Values

This function returns

- 0 on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_CERR_MEMORY	Insufficient memory to queue the channel
WIP_CERR_INVALID	NULL channel specified

Finalizers

Channels generally reserve some heap memory. Depending on their features, it can take some time between the call to wip_close function and the actual releasing of the resources. Although the user might be interested into knowing when a channel closing procedure has been completed, it cannot be reported as a wip_event_t. Since Internet Library events are attached to the channel, and by definition, the channel does not exist after its release.The users should not use a wip_channel_t in any way after wip_close function has been called on it. If they do, unspecified problems including reboot and memory corruptions might occur. So in order to let users monitor the completion of a channel closure, most channels can be added with a finalizer.

A finalizer is a function which is called after the channel has been completely closed and all its associated resources are freed. Finalizers are attached to channels with the WIP_COPT_FINALIZER option, either in the wip_xxxCreateOpts function, or in the wip_setOpts function.This option will allow to pass a finalizer function to the channel. Refer to the section 6.6.3.2 for details about the option.

```
typedef void (*wip_finalizer_f) (void *ctx);
```

This callback type is wip_finalizer_f, and takes the following as parameters:

ctx:

context argument which was attached to the channel.

Note: It is illegal to try to access the (recently destroyed) channel in the finalizer.

Please refer to the Simple Finalizer Example section for a detailed example on the usage of finalizer function.

6.2.2. The wip_read Function

The wip_read function is used to read from a channel. For more details see the Options overview section.

Prototype

```
int wip_read ( wip_channel_t   c,
               void           *buffer,
               u32             buf_len );
```

Parameters

c:

In: The channel to read from

buffer:

Out: Pointer to the buffer where read data must be put

buf_len:

In: Size of the buffer

Returned Values

This function returns

- number of bytes actually read on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_CERR_CSTATE	The channel is not ready to read data (still in initialization, or is already closed).
WIP_CERR_NOT_SUPPORTED	This channel does not support data reading.

Note: *wip_read function returns zero when there is no more data to read from the channel. In this case the application should wait for WIP_CEV_READ event before invoking wip_read function again.*

6.2.3. The wip_readOpts Function

The wip_readOpts function is used to read from a channel. For more details see the Options overview section.

Prototype

```
int wip_readOpts ( wip_channel_t  c,
                   void          *buffer,
                   u32            buf_len,
                                  ... );
```

Parameters

c:

In: The channel to read from

buffer:

Out: Pointer to the buffer where read data must be put

buf_len:

In: Size of the buffer

...:

List of option names followed by option values. The list must be terminated by WIP_COPT_END. Supported options depend on the kind of channel and are mentioned in the TCP wip_readOpts Function and UDP wip_readOpts Function sections.

Returned Values

This function returns:

- number of bytes actually read
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_CERR_CSTATE	The channel is not ready to read data (still in initialization, or is already closed)
WIP_CERR_NOT_SUPPORTED	This channel does not support data reading, or it has been provided with an option it does not support.
WIP_CERR_INVALID	Some option has been passed with an invalid value.

6.2.4. The wip_write Function

The wip_write function is used to write to a channel. For more details see the Options overview section.

Prototype

```
int wip_write ( wip_channel_t   c,
                void           *buffer,
                u32             buf_len );
```

Parameters

c:

In: The channel to write to

buffer:

Out: Pointer to the buffer where data to write is to be found

buf_len:

In: Size of the buffer

Returned Values

This function returns

- number of bytes actually written
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_CERR_CSTATE	The channel is not ready to write data (still in initialization, or is already closed).
WIP_CERR_NOT_SUPPORTED	This channel does not support data writing.

Note: *When return value of wip_write function is zero or write attempt writes less data than the requested data, then an application must wait for WIP_CEV_WRITE event before calling wip_write function again.*

6.2.5. The wip_writeOpts Function

The wip_writeOpts function is used to write to a channel. For more details see the Options overview section.

Prototype

```
int wip_writeOpts ( wip_channel_t   c,
                    void   *buffer,
                    u32   buf_len,
                    ... );
```

Parameters

c:

In: The channel to write to

buffer:

Out: Pointer to the buffer where data to be written can be found

buf_len:

In: Size of the buffer

...:

List of option names followed by option values. The list must be terminated by WIP_COPT_END. Supported options depend on the kind of channel and are mentioned in the TCP wip_writeOpts Function and UDP wip_writeOpts Function sections.

Returned Values

This function returns

- number of bytes actually written
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_CERR_CSTATE	The channel is not ready to write data (still in initialization, or is already closed)
WIP_CERR_NOT_SUPPORTED	This channel does not support data writing, or it has been provided with an option it does not support.
WIP_CERR_INVALID	Some option has been passed with an invalid value.

6.2.6. The wip_getOpts Function

The wip_getOpts function is used to get options from a channel. For more details see the Options overview section.

Note: Socket/Session should be active to retrieve option values using wip_getOpts ().

Prototype

```
int wip_getOpts ( wip_channel_t   c,
                  ... );
```

Parameters

c:

In: The channel to get options from

...:

List of option names followed by option values. The list must be terminated by WIP_COPT_END. Supported options depend on the kind of channel and are mentioned in sections 6.3.4, 6.4.3, 6.5.7, 6.6.3, 8.5, 9.15, 10.3.2 and 11.2.2.

Returned Values

This function returns

- zero on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_CERR_NOT_SUPPORTED	The function has been provided with an option it does not support.
WIP_CERR_INVALID	Some option has been passed with an invalid value.
WIP_CERR_CSTATE	The channel is not ready to get options (still in initialization, or is already closed)

6.2.7. The wip_setOpts Function

The wip_setOpts function is used to set options for a channel. For more details see the Options overview section.

Note: Socket/Session should be active to change option values using wip_setOpts ().

Prototype

```
int wip_setOpts ( wip_channel_t   c,
                                  ... );
```

Parameters

c:

In: The channel in which options will be set

...:

List of option names followed by option values. The list must be terminated by WIP_COPT_END. Supported options depend on the kind of channel and are mentioned in sections 6.3.5, 6.4.4, 6.5.8, 6.6.4, 8.4 and 9.14.

Returned Values

This function returns

- zero on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_CERR_NOT_SUPPORTED	The function has been provided with an option it does not support.
WIP_CERR_INVALID	Some option has been passed with an invalid value.

6.2.8. The wip_setCtx Function

The wip_setCtx function is used to change the context associated with the event handler of a channel.

Prototype

```
void wip_setCtx ( wip_channel_t   c,
                  void            *ctx );
```

Parameters

c:

The channel for which the event context must be changed

ctx:

The new context

Returned Values

None

6.2.9. The wip_getState Function

Channel creation might rely on asynchronous processes such as the completion of DNS query. There is therefore no guarantee that immediately after the wip_xxxCreate function returns, the channel is ready for read/write operations. Moreover, some events, especially errors, can put a channel in an unusable state. These different states are summarized by the wip_cstate_t enumeration, and the current state of a channel can be read with wip_getState.

Prototype

```
wip_cstate_t wip_getState ( wip_channel_t   c );
```

Parameter

c:

The channel for which the state must be determined

Returned Values

This function returns the state of c as one of the values below:

```
typedef enum wip_cstate_t {
  WIP_CSTATE_BUSY,                     /* some configuration is happening,
                                          eventually the state will become
                                          READY*/
  WIP_CSTATE_READY,                    /* Ready to support operations.*/
  WIP_CSTATE_TO_CLOSE,                 /* Channel is broken; the only thing
                                          to do is to close it.*/

  WIP_CSTATE_LAST = WIP_CSTATE_TO_CLOSE

} wip_cstate_t;
```

6.3. UDP: UDP Sockets

UDP sockets are not connected; they do not have a peer socket with which they exclusively exchange data. However, as in POSIX sockets, we offer a pseudo-connected optional API. The user can specify a destination socket, to which every outbound packet will be sent through a given socket, until further notice. If no pseudo-connection is established, it is mandatory to specify the destination address and port for every write operation, through WIP_COPT_XXX options; therefore, a call to wip_write() on an unconnected UDP will fail.

Note: Access to the UDP inside its private network is blocked by some of the service provider. In this case, contact service provider to get special settings to connect to the UDP server.

6.3.1. State Charts

The functional behavior of UDP sockets is formalized on the following statechart. The green background label represents events, and the blue background represents functions called by the user.

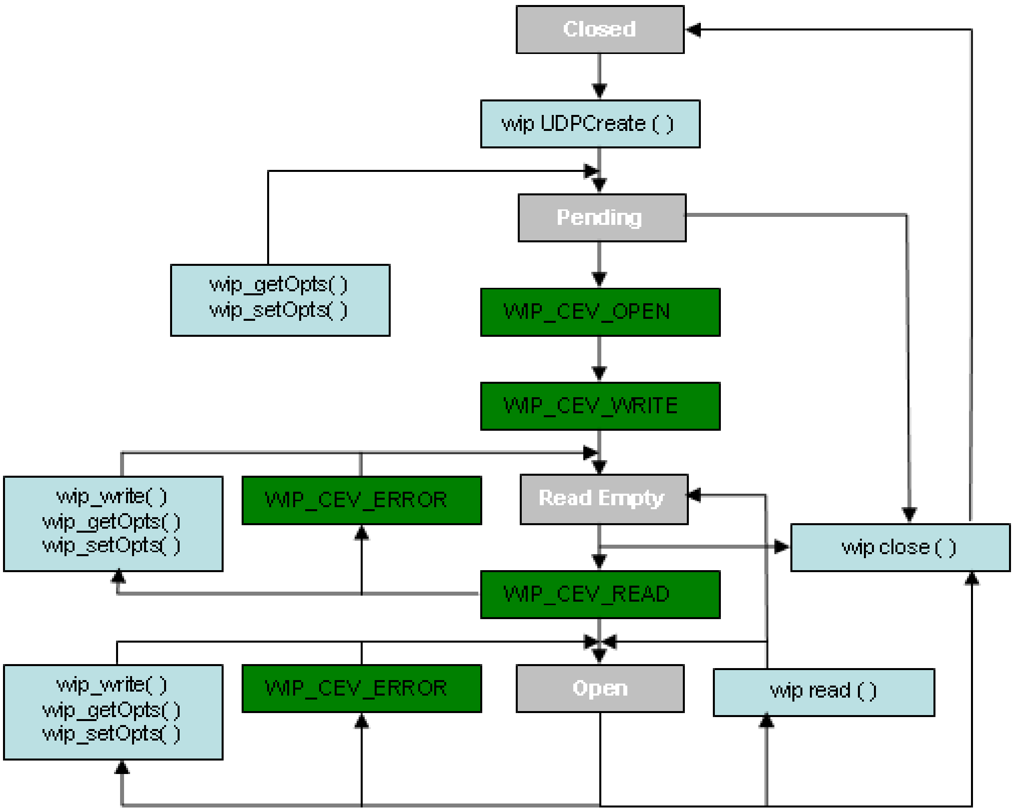


Figure 10. UDP Channel State Diagram

A more intuitive example of temporal dataflow, inferred from this state diagram is given below. It shows typical UDP channels opening, data transfers between sockets, and channel closing.

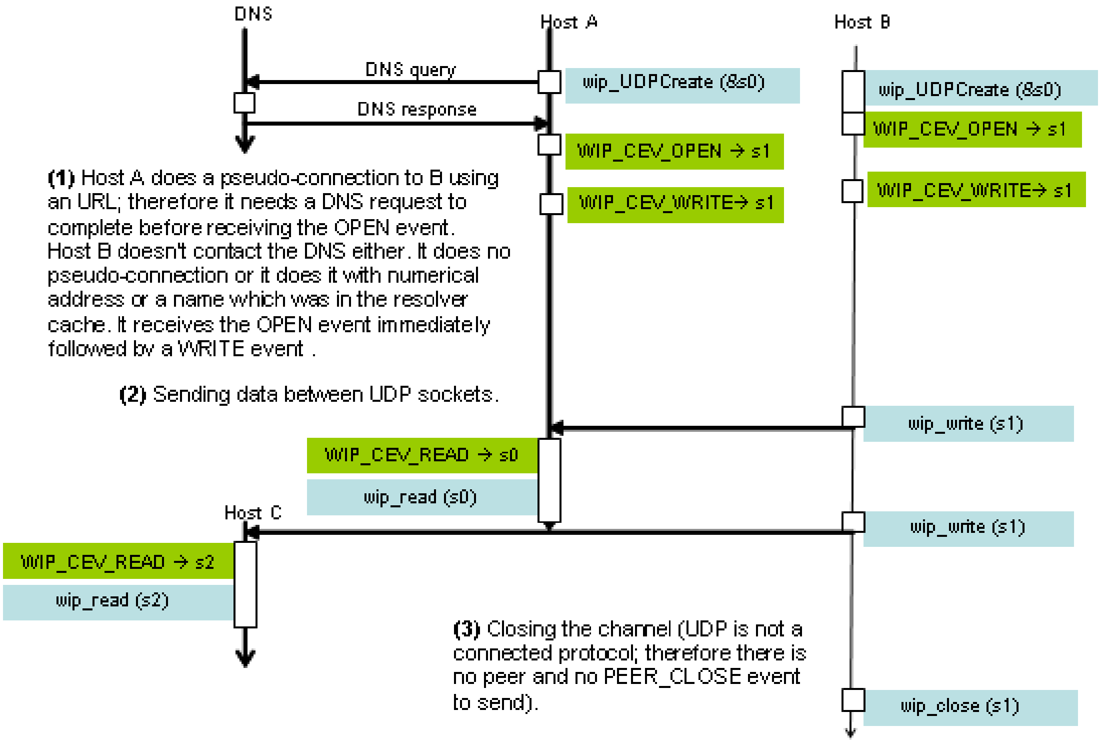


Figure 11. UDP Channel Temporal Diagram

6.3.2. The wip_UDPCreate Function

The wip_UDPCreate function creates a channel encapsulating an UDP socket.

Prototype

```
wip_channel_t wip_UDPCreate ( wip_eventHandler_f   evHandler,

                              void   *ctx );
```

Parameters

`evHandler:`

The call back handler which receives the network events related to the UDP socket. Possible events kinds are WIP_CEV_READ, WIP_CEV_WRITE and WIP_CEV_ERROR. If set to NULL, all the events received in this socket will be discarded.

`ctx:`

User data to be passed to the event handler every time it is called

Returned Values

This function returns

- the created channel
- NULL on error

6.3.3. The wip_UDPCreateOpts Function

The wip_UDPCreateOpts function creates a channel encapsulating an UDP socket, with advanced options.

Prototype

```
wip_channel_t wip_UDPCreateOpts ( wip_eventHandler_f   evHandler,
                                  void   *ctx,
                                  ... );
```

Parameters

evHandler:

The call back handler which receives the network events related to the UDP socket. Possible event kinds are WIP_CEV_READ, WIP_CEV_WRITE and WIP_CEV_ERROR. If set to NULL, all events received in this socket will be discarded.

ctx:

User data to be passed to the event handler every time it is called

...:

List of option names followed by option values. The list must be terminated by WIP_COPT_END. The supported options are:

Option	Value	Description
WIP_COPT_SND_BUFSIZE	u32	Size of the emission buffer associated with a socket. default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32	Size of the reception buffer associated with a socket. default: depends on the protocol.
WIP_COPT_CHECKSUM	bool	Whether the checksum control must be performed by an UDP socket. default:TRUE
WIP_COPT_TOS	u8	Type of Service (cf. RFC 791) default:0
WIP_COPT_TTL	u8	Time-To-Live for packets. default:64
WIP_COPT_DONTFRAG	bool	If set. UDP datagrams are not allowed to be fragmented when going through the network. default:FALSE
WIP_COPT_PORT	u16	Port occupied by this socket. default:0
WIP_COPT_STRADDR	ascii*	Local address of the socket. default:0
WIP_COPT_ADDR	wip_in_addr_t	Local address of the socket. default:0

Option	Value	Description
WIP_COPT_PEER_PORT	u16	Port of the peer socket. default:0
WIP_COPT_PEER_STRADDR	ascii*	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection. default:0
WIP_COPT_PEER_ADDR	wip_in_addr_t	Address of the peer socket. default:0
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.
WIP_COPT_BOUND	boolean	Specifies whether the socket is bound2 to a peer socket or not. default: 1

2 The option WIP_COPT_BOUND is used to check whether an UDP socket is bound to any other UDP socket or not. When the UDP socket is created without specifying the IP address of the peer, then the option WIP_COPT_BOUND will be read as FALSE. This is because there is no destination IP address to communicate with. If the UDP socket is created by specifying the peer IP address, the option WIP_COPT_BOUND will be read as TRUE. This is because the peer IP address will be resolved by the DNS and the socket is said to be bounded to the peer socket. Hence this option will be read as TRUE.

Returned Values

This function returns

- the created channel
- NULL on error

6.3.4. The wip_getOpts Function

The options supported by the wip_getOpts function, applied to a UDP are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_SND_BUFSIZE	u32*	Size of the emission buffer associated with a socket default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32*	Size of the reception buffer associated with a socket default: depends on the protocol.
WIP_COPT_ERROR	s32*	Number of the last error experienced by that socket. default:WM_EOK
WIP_COPT_NREAD	u32*	Number of bytes that can currently be read on that socket. default:0
WIP_COPT_NWRITE	u32*	Number of bytes that can currently be written on that socket. For a PING, size of the request default:20
WIP_COPT_CHECKSUM	bool*	Whether the checksum control must be performed by an UDP socket. default:TRUE
WIP_COPT_TOS	u8*	Type of Service (cf. RFC 791) default:0
WIP_COPT_TTL	u8*	Time-To-Live for packets. default:64
WIP_COPT_DONTFRAG	bool*	If set. UDP datagrams are not allowed to be fragmented when going through the network. default:FALSE
WIP_COPT_PORT	u16*	Port occupied by this socket. default:0
WIP_COPT_STRADDR	ascii* buffer, u32 buf_len	Local address of the socket. default:0
WIP_COPT_ADDR	wip_in_addr_t*	Local address of the socket, as a 32 bits integer. default:0
WIP_COPT_PEER_PORT	u16*	Port of the peer socket. default:0
WIP_COPT_PEER_STRADDR	ascii* buff, u32 buf_len	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection. default:0
WIP_COPT_PEER_ADDR	wip_in_addr_t*	Address of the peer socket, as a 32 bits integer. default:0
WIP_COPT_SUPPORT_READ	none	Fails if the channel does not support wip_read() operations. If supported, does nothing.
WIP_COPT_SUPPORT_WRITE	none	Fails if the channel does not support wip_write() operations. If supported, does nothing.
WIP_COPT_BOUND	boolean	Specifies whether the socket is bound2 to a peer socket or not. default:1

2 The option WIP_COPT_BOUND is used to check whether an UDP socket is bound to any other UDP socket or not. When the UDP socket is created without specifying the IP address of the peer, then the option WIP_COPT_BOUND will be read as FALSE. This is because there is no destination IP address

to communicate with. If the UDP socket is created by specifying the peer IP address, the option WIP_COPT_BOUND will be read as TRUE. This is because the peer IP address will be resolved by the DNS and the socket is said to be bounded to the peer socket. Hence this option will be read as TRUE.

6.3.5. The wip_setOpts Function

The options supported by the wip_setOpts function, applied to a UDP are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_SND_BUFSIZE	u32	Size of the emission buffer associated with a socket. default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32	Size of the reception buffer associated with a socket. default: depends on the protocol.
WIP_COPT_CHECKSUM	bool	Whether the checksum control must be performed by an UDP socket. default:TRUE
WIP_COPT_TOS	u8	Type of Service (cf. RFC 791) default:0
WIP_COPT_TTL	u8	Time-To-Live for packets. default:64
WIP_COPT_DONTFRAG	bool	If set. UDP datagrams are not allowed to be fragmented when going through the network. default:FALSE
WIP_COPT_PEER_PORT	u16	Port of the peer socket. default:0
WIP_COPT_PEER_STRADDR	ascii*	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection. default:0
WIP_COPT_PEER_ADDR	wip_in_addr_t	Address of the peer socket, as a 32 bits integer. default:0
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

Note: The range of values for the WIP_COPT_SND_BUFSIZE and WIP_COPT_RCV_BUFSIZE options is 1-65535.

6.3.6. The wip_readOpts Function

The options supported by the wip_readOpts function, applied to a UDP are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_PEEK	bool	When true, the message is not deleted from the buffer after reading, so that it can be read again. default:FALSE
WIP_COPT_PEER_PORT	u16*	Port of the peer socket. default:0
WIP_COPT_PEER_STRADDR	ascii *buffer, u32 buf_len	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection. default:0
WIP_COPT_PEER_ADDR	wip_in_addr_t*	Address of the peer socket, as a 32 bits integer. default:0

6.3.7. The wip_writeOpts Function

The options supported by the wip_writeOpts function, applied to a UDP are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_PEER_PORT	u16*	Port of the peer socket. default:0
WIP_COPT_PEER_STRADDR	ascii *buffer, u32 buf_len	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection. default:0
WIP_COPT_PEER_ADDR	wip_in_addr_t*	Address of the peer socket, as a 32 bits integer. default:0

6.4. TCPServer: Server TCP Sockets

TCP server sockets do not support direct data communications. Instead, they spawn new TCPClient TCP communication sockets whenever a peer socket requests a connection. They do not have a meaningful event handler, as they cannot be closed (they have no peer socket) and cannot experience an error once they have been successfully created.

Note: Access to the TCP server inside its private network is blocked by some of the service provider. In this case, contact service provider to get special settings to connect to the TCP server.

The state diagram is as follows:

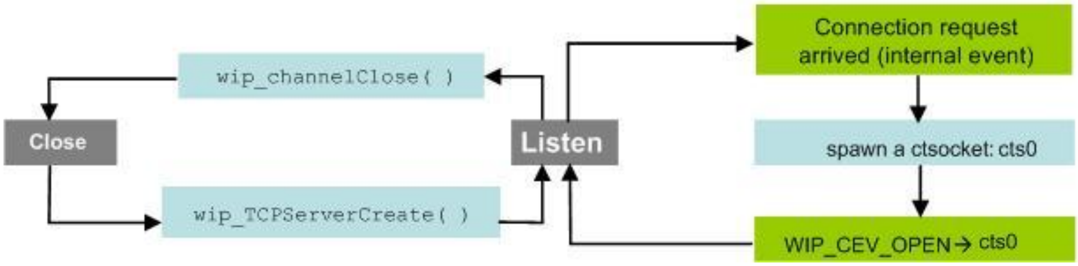


Figure 12. TCP Server Channel State Diagram

There is no relevant temporal diagram to give here. Once the server socket is created, the only direct interaction the user can have with it is by closing it. Reacting to communication socket spawning is done by handling the WIP_CEV_OPEN events of the spawned sockets.

6.4.1. The wip_TCPServerCreate Function

The wip_TCPServerCreate function creates a channel encapsulating a TCP server socket.

Prototype

```
wip_channel_t wip_TCPServerCreate ( u16   port,
                                    wip_eventHandler_f   spawnedHandler,
                                    void   *ctx );
```

Parameters

port:

The port number on which TCP server socket listens

spawnedHandler:

The call back handler which receives the events related to the TCP clients. It is important to realize that this handler will react to events happening to the resulting communication sockets, not to those happening to the server socket. The context initially linked with this handler is ctx, although it can be later changed, on a per-TCP client basis, through wip_setCtx().

ctx:

User data passed to the event handlers of the spawned sockets

Returned Values

This function returns

- the created channel
- NULL on error

6.4.2. The wip_TCPServerCreateOpts Function

The wip_TCPServerCreateOpts function creates a channel encapsulating a TCP server socket with user defined settings.

Prototype

```
wip_channel_t wip_TCPServerCreateOpts ( u16   port,

                                        wip_eventHandler_f
spawnedHandler,

                                        void   *ctx,

                                        ... );
```

Parameters

port:

The port number on which TCP server socket listens

spawnedHandler:

The call back handler which receives the events related to the TCP clients. It is important to realize that this handler will react to events happening to the resulting communication sockets, not to those happening to the server socket. The context initially linked with this handler is ctx, although it can be later changed, on a per-TCPClient basis, through wip_setCtx().

ctx:

User data passed to the event handlers of the spawned sockets

...:

Same as wip_TCPServerCreate(), plus a list of option names must be followed by option values. The list must be terminated by WIP_COPT_END. The options supported by wip_TCPServerCreateOpts() are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_SND_BUFSIZE	u32 (inherited by spawned TCPClients)	Size of the emission buffer associated with a socket. default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32 (inherited by spawned TCPClients)	Size of the reception buffer associated with a socket. default: depends on the protocol.
WIP_COPT_SND_LOWAT	u32 (inherited by spawned TCPClients)	Minimum amount of available space that must be available in the emission buffer before triggering a WIP_CEV_WRITE event. default:1024
WIP_COPT_RCV_LOWAT	u32 (inherited by spawned TCPClients)	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ event. default:1

Option	Value	Description
WIP_COPT_NODELAY	bool (inherited by spawned TCPClients)	When set to TRUE, TCP packets are sent immediately, even if the buffer is not full enough When set to FALSE, the packets will be sent either, a) by combining several small packets into a bigger packet b) when the data is ready to send and the stack is idle Note: Data has to be buffered and managed by the user application. There is no provision in Internet Library APIs to wait for data block to be fully filled before sending it. default:FALSE
WIP_COPT_TOS	u8 (inherited by spawned TCPClients)	Type of Service (cf. RFC 791) default:0
WIP_COPT_TTL	u8 (inherited by spawned TCPClients)	Time-To-Live for packets sent. default:64
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.
WIP_COPT_REXMT_MAX	u32 (inherited by spawned TCPClients)	Sets the maximum time between TCP retransmissions. default:64 seconds
WIP_COPT_REXMT_MAXCNT	u32 (inherited by spawned TCPClients)	Sets the maximum number of retransmissions. default:12

Most of these options are inherited by spawned TCPClients. That is, they have no effect on the TCPServer itself, but when the TCPServer creates new TCPClients through an accept function call, these TCPClients are initialized with those options.

Note: The range of values for the WIP_COPT_REXMT_MAX option is the range of value coded on an u32 and the range of value for WIP_COPT_REXMT_MAXCNT option is 0-12.

Returned Values

This function returns

- the created channel
- NULL on error

6.4.3. The wip_getOpts Function

The options supported by the wip_getOpts function, applied to a TCPServer are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_KEEPALIVE	u32* n	Sends a NOOP command every n tenth of seconds, so that the server and any NAT on the way won't shut down the connection default:1
WIP_COPT_SND_BUFSIZE	u32*	Size of the emission buffer associated with a socket. default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32*	Size of the reception buffer associated with a socket. default: depends on the protocol.
WIP_COPT_SND_LOWAT	u32*	Minimum amount of available space that must be available in the emission buffer before triggering a WIP_CEV_WRITE event. default:1024
WIP_COPT_RCV_LOWAT	u32*	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ event. default:1
WIP_COPT_NODELAY	bool*	When set to TRUE, TCP packets are sent immediately, even if the buffer is not full enough When set to FALSE, the packets will be sent either, a) by combining several small packets into a bigger packet b) when the data is ready to send and the stack is idle Note: Data has to be buffered and managed by the user application. There is no provision in Internet Library APIs to wait for data block to be fully filled before sending it. default:FALSE
WIP_COPT_TOS	u8*	Type of Service (cf. RFC 791) default:0
WIP_COPT_TTL	u8*	Time-To-Live for packets sent through this socket; Time-To-Live for this packet, when used in a wip_writeOpts(). default:64
WIP_COPT_PORT	u16*	Port occupied by this socket. default:0
WIP_COPT_STRADDR	ascii* buff, u32 buf_len	Local address of the socket. default:0
WIP_COPT_ADDR	wip_in_addr_t*	Local address of the socket, as a 32 bits integer. default:0
WIP_COPT_REXMT_MAX	u32*	Gets the maximum time between TCP retransmissions. default:64 seconds
WIP_COPT_REXMT_MAXCNT	u32*	Gets the maximum number of retransmissions. default:12

6.4.4. The wip_setOpts Function

The options supported by the wip_setOpts function, applied to a TCPServer are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_KEEPALIVE	u32 n	Sends a NOOP command every n tenth of seconds, so that the server and any NAT on the way won't shut down the connection default:1
WIP_COPT_SND_BUFSIZE	u32 (inherited by spawned TCPClients)	Size of the emission buffer associated with a socket. default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32 (inherited by spawned TCPClients)	Size of the reception buffer associated with a socket. default: depends on the protocol.
WIP_COPT_SND_LOWAT	u32 (inherited by spawned TCPClients)	Minimum amount of available space that must be available in the emission buffer before triggering a WIP_CEV_WRITE event. default:1024
WIP_COPT_RCV_LOWAT	u32 (inherited by spawned TCPClients)	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ event. default:1
WIP_COPT_NODELAY	bool (inherited by spawned TCPClients)	When set to TRUE, TCP packets are sent immediately, even if the buffer is not full enough When set to FALSE, the packets will be sent either, a) by combining several small packets into a bigger packet b) when the data is ready to send and the stack is idle Note: Data has to be buffered and managed by the user application. There is no provision in Internet Library APIs to wait for data block to be fully filled before sending it default:FALSE
WIP_COPT_TOS	u8 (inherited by spawned TCPClients)	Type of Service (cf. RFC 791) default:0
WIP_COPT_TTL	u8 (inherited by spawned TCPClients)	Time-To-Live for packets. default:64
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.
WIP_COPT_REXMT_MAX	u32	Sets the maximum time between TCP retransmissions. default:64 seconds
WIP_COPT_REXMT_MAXCNT	u32	Sets the maximum number of retransmissions. default:12

Note: *WIP_COPT_SND_BUFSIZE and WIP_COPT_RCV_BUFSIZE can be set to 0. For instance, if user always wants to send data and not to receive any incoming data, then it will be useful to set socket read buffer size to zero, to save memory.*

The range of values for the WIP_COPT_REXMT_MAX option is the range of value coded on an u32 and the range of value for WIP_COPT_REXMT_MAXCNT option is 0-12.

6.5. TCPClient: TCP Communication Sockets

Communication TCP sockets, can either be created as client TCP sockets, or spawned by a server TCP socket. Although there are two distinct ways to create communication sockets, on client-side and server-side, once they are created and connected together, they are symmetrical and share the same API.

6.5.1. Read/Write Events

Read Events

READ event will be received:

- first time if there is more than WIP_COPT_RCV_LOWAT bytes to read in the socket's read buffer
- when read attempt returns less data than the requested data and there is more than WIP_COPT_RCV_LOWAT bytes available in the buffer

Let's consider an example,

WIP_COPT_RCV_BUFSIZE (MAX) has been set to 5840 bytes and WIP_COPT_RCV_LOWAT (MIN) has been set to 1000 bytes.

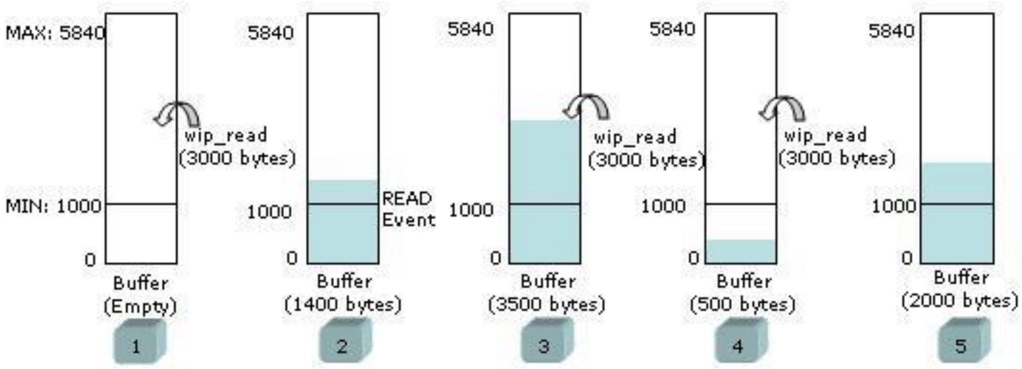


Figure 13. Generation of Read event

In this example, the diagram shown above explains the scenario when READ events are received:

Step 1: Attempt is made to read data (3000 bytes).The buffer is empty as data has not been received, so no READ event is received and read will fail.

Step 2: Received 1400 bytes of data in the buffer. In this case, READ event will be received as the size of readable data in the buffer is more than WIP_COPT_RCV_LOWAT, and no READ event has been sent since the last unsuccessful attempt to read.

Step 3: More data (2100 bytes) is received in the buffer. In this case, READ event will not be received, as READ event was already received in Step 2. Data is read (3000 bytes) from the buffer. Size of readable data in the buffer is 500 bytes.

Step 4: Data is read (1500 bytes) from the buffer. Read attempt reads (500 bytes) less data than the requested data, as the available data in the buffer is less.

Step 5: More data (1500 bytes) is received in the buffer. In this case, since the size of the readable data in the buffer (2000 bytes) is more than WIP_COPT_RCV_LOWAT, and there has been an incomplete read (at step 4) since last time a READ event has been received, a new READ event will be received.

Note: The dgm_size field in the event is not set when a READ event occurs. It will not be reliable, because the amount of readable data might change when new data arrives between when the event is generated, and when it is processed by the application. dgm_size is only applicable for datagram-oriented protocols

No READ event will be received when data is read from the buffer and the size of readable data is more than WIP_COPT_RCV_LOWAT and more data is received.

Write Events

WRITE event will be received when:

- channel is opened for the first time
- write attempt writes less data than the requested data and there are more than WIP_COPT_SND_LOWAT bytes available in the buffer

Let's consider an example,

WIP_COPT_SND_BUFSIZE (MAX) has been set to 5840 bytes and WIP_COPT_SND_LOWAT (MIN) has been set to 1000 bytes.

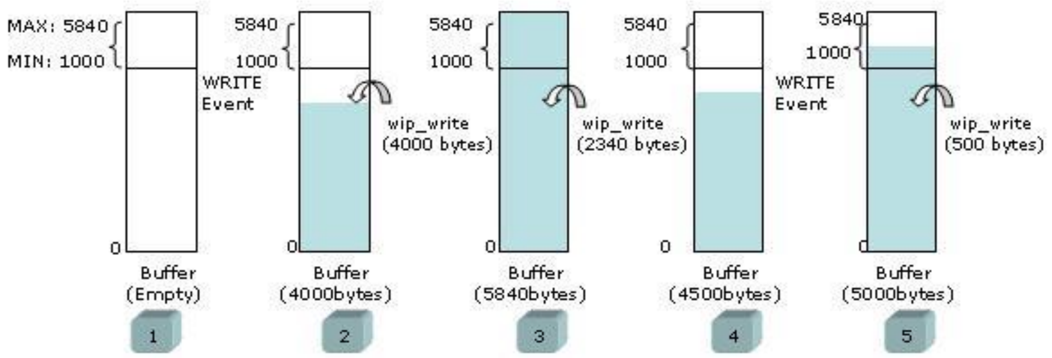


Figure 14. Generation of Write event

In this example, the diagram shown above explains the scenario when WRITE events are received:

Step 1: WRITE event is received as the channel is opened for the first time and the buffer is empty.

Step 2: 4000 bytes of data are written to the buffer. In this case, WRITE event will not be received as there is still memory (1840 bytes) to write more data

Step 3: Attempt is made to write data (2340 bytes) more than available buffer size. In this case, only 1840 bytes of data is written successfully to the buffer as the free buffer size is 1840 bytes. Remaining data (500 bytes) will be written to the buffer when the free buffer size becomes equal or more than WIP_COPT_SND_LOWAT.

Step 4: Data is flushed (1340 bytes) from the buffer and now the free buffer is 1340 bytes. In this case, WRITE event will be received, as the free buffer is more than WIP_COPT_SND_LOWAT and there has been no WRITE event since last time a WRITE event has been received.

Step 5: Remaining data (500 bytes) is written to the buffer. In this case, WRITE event will not be received, as there is still memory (840 bytes) to write more data.

6.5.2. Statecharts

The complete state diagram of a TCP communication socket is given below:

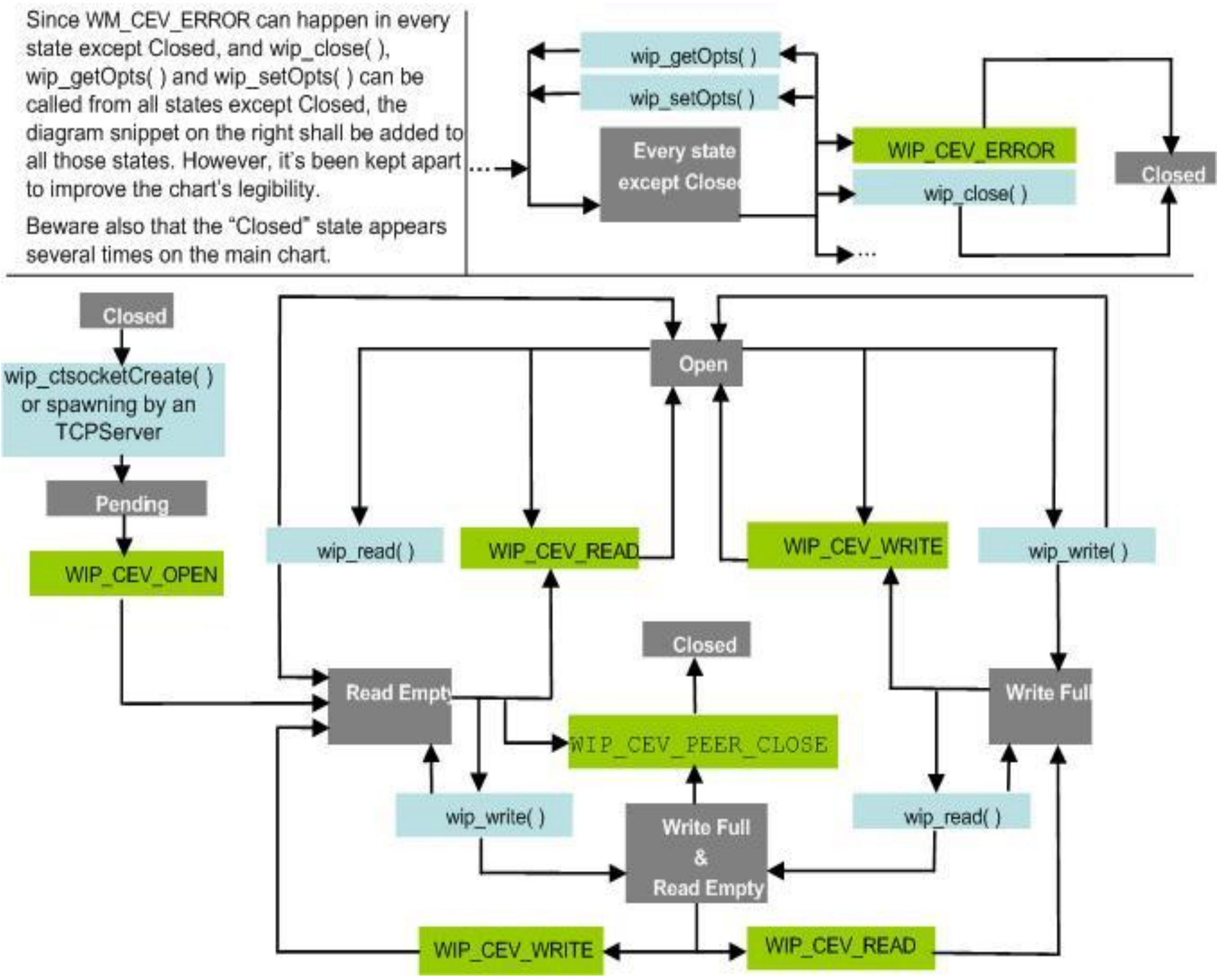


Figure 15. TCP Communication Channel State Diagram

This state diagram might be considered too complex for practical reference. The "OpenReady", "Read empty", "Write full", "Write full and Read empty" states can be unified. The resulting state diagram will be simpler, but will not predict whether non-blocking read/write operations will succeed. It does not precisely specify when the WIP_CEV_READ, WIP_CEV_WRITE and WIP_CEV_PEER_CLOSE events can occur.

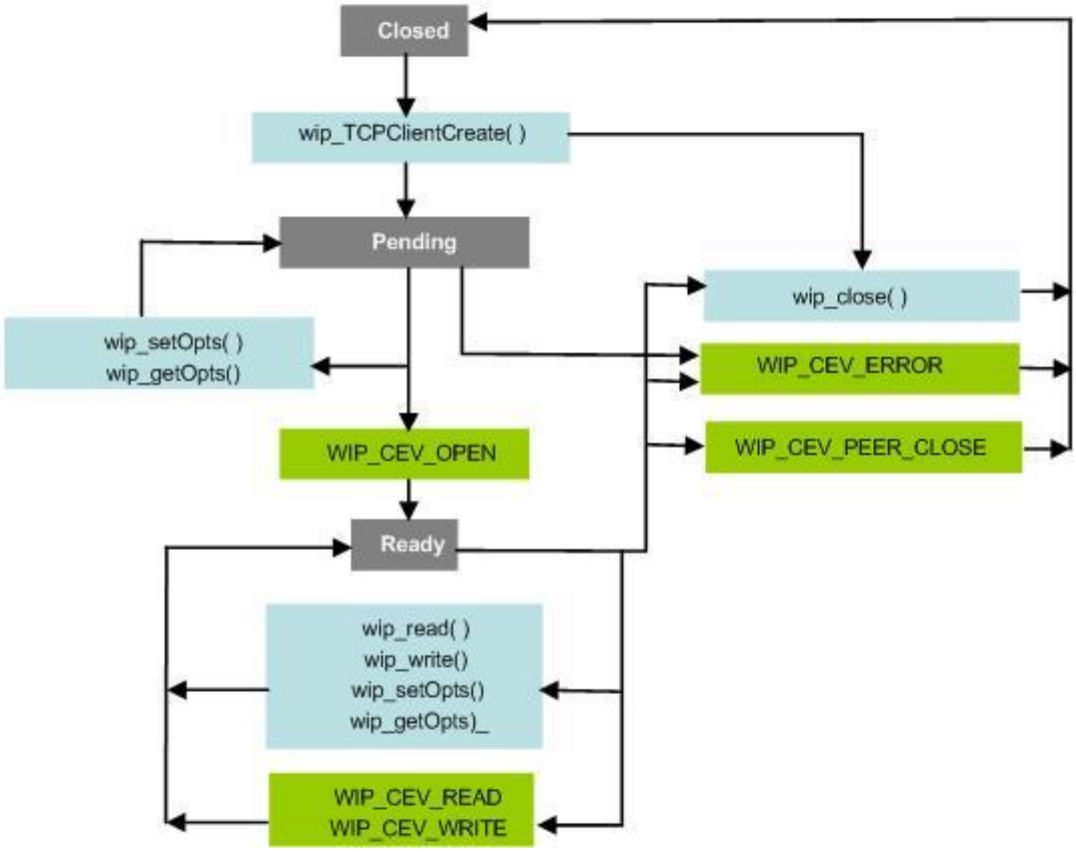


Figure 16. TCP Communication Channel Simplified State Diagram

A typical temporal flow example follows:

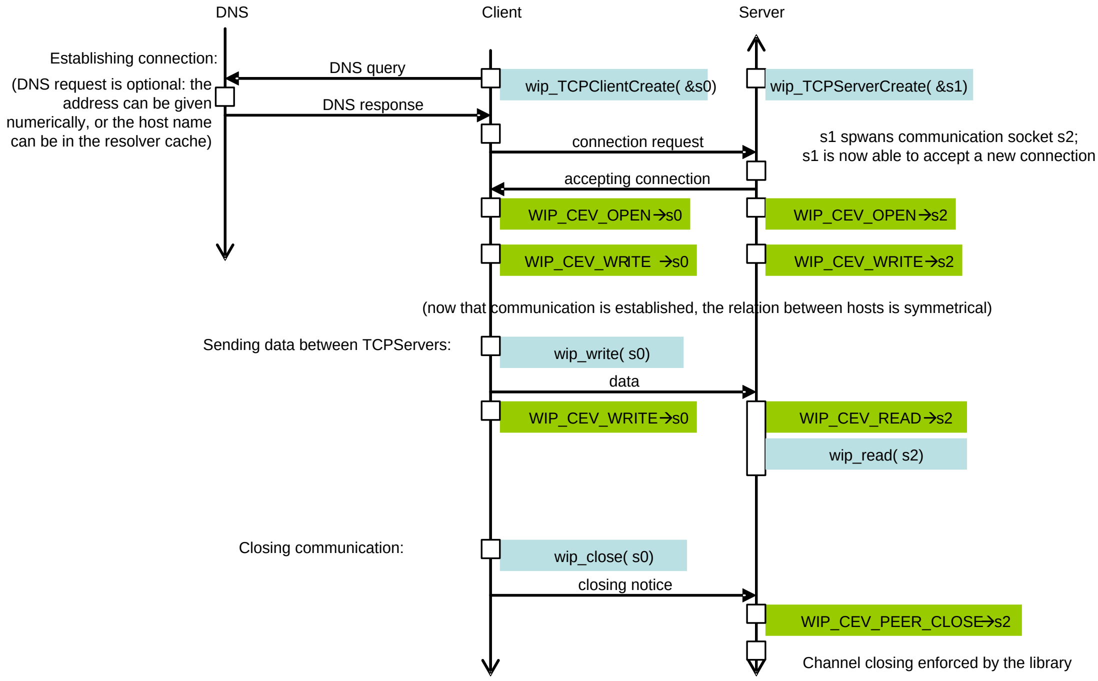


Figure 17. TCP Communication Channel Temporal Diagram

6.5.3. The wip_TCPClientCreate Function

The wip_TCPClientCreate function creates a channel encapsulating a TCP client socket.

Prototype

```
wip_channel_t wip_TCPClientCreate ( const ascii   *serverAddr,
                                    u16   serverPort,
                                    wip_eventHandler_f   evHandler,
                                    void   *ctx );
```

Parameters

serverAddr:

Address of the destination server which can be either a DNS address, or a numeric one in the form “xxx.xxx.xxx.xxx”.

serverPort:

Port of the server socket to connect to

evHandler:

The call back handler which receives the network events related to the socket. Possible events kinds are WIP_CEV_READ, WIP_CEV_WRITE, WIP_CEV_PEER_CLOSE and WIP_CEV_ERROR. If set to NULL, all events received in this socket will be discarded.

ctx:

User data to be passed to the event handler every time it is called

Returned Values

This function returns

- the created channel
- NULL on error

6.5.4. The wip_TCPClientCreateOpts Function

The wip_TCPClientCreateOpts function creates a channel encapsulating a TCP client socket, with advanced options.

Prototype

```
wip_channel_t wip_TCPClientCreateOpts ( const ascii   *serverAddr,
                                        u16   serverPort,
                                        wip_eventHandler_f   evHandler,
                                        void   *ctx,
                                        ... );
```

Parameters

The parameters are the same as the parameters for the wip_TCPClientCreate() function, plus list of option names. The list of option names must be followed by option values. The list must be terminated by WIP_COPT_END .The options supported by wip_TCPServerCreateOpts() are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_KEEPALIVE	u32 n	Sends a NOOP command every n tenth of seconds, so that the server and any NAT on the way won't shut down the connection default:1
WIP_COPT_SND_BUFSIZE	u32	Size of the emission buffer associated with a socket. default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32	Size of the reception buffer associated with a socket. default: depends on the protocol.
WIP_COPT_SND_LOWAT	u32	Minimum amount of available space that must be available in the emission buffer before triggering a WIP_CEV_WRITE event. default:1024
WIP_COPT_RCV_LOWAT	u32	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ event. default:1
WIP_COPT_NODELAY	bool	When set to TRUE, TCP packets are sent immediately, even if the buffer is not full enough When set to FALSE, the packets will be sent either, a) by combining several small packets into a bigger packet b) when the data is ready to send and the stack is idle Note: Data has to be buffered and managed by the user application. There is no provision in Internet Library APIs to wait for data block to be fully filled before sending it. default:FALSE

Option	Value	Description
WIP_COPT_MAXSEG	u32	Maximum size of TCP packets
WIP_COPT_TOS	u8	Type of Service (cf. RFC 791) default:0
WIP_COPT_TTL	u8	Time-To-Live for packets. default:64
WIP_COPT_STRADDR	ascii*	Local address of the socket. default:0
WIP_COPT_PORT	u16	Port occupied by this socket. default:0
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.
WIP_COPT_REXMT_MAX	u32	Sets/gets the maximum time between TCP retransmissions. default:64 seconds
WIP_COPT_REXMT_MAXCNT	u32	Sets/gets the maximum number of retransmissions. default:12

Note: The range of values for the WIP_COPT_REXMT_MAX option is the range of value coded on an u32 and the range of value for WIP_COPT_REXMT_MAXCNT option is 0-12.

Returned Values

This function returns

- the created channel
- NULL on error

6.5.5. The wip_abort Function

The wip_abort function aborts a TCP communication, causing an error on the peer socket.

Note: *wip_abort () can only be used to abort a valid or a non-closed channel. The behavior of this API is not defined, if it is used to abort a non-created or a closed channel.*

Prototype

```
int wip_abort ( wip_channel_t   c );
```

Parameters

c:

The socket that must be aborted

Returned Values

This function returns

- zero on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_CERR_NOT_SUPPORTED	Returned when abort is requested on TCP server or UDP channels
WIP_CERR_INTERNAL	Impossible to abort the TCP communication due to internal reasons

6.5.6. The wip_shutdown Function

The wip_shutdown function shuts down input and/or output communication on the socket. If both communications are shut down, the socket is closed. If the output communication is closed, the peer socket receives by a WIP_CEV_PEER_CLOSE error event.

Prototype

```
int wip_shutdown ( wip_channel_t   c,
                   bool   read,
                   bool   write );
```

Parameters

c:

The socket that must be shut down

read:

Whether the input communication must be shut down

write:

Whether the output communication must be shut down

Returned Values

This function returns

- zero on success
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_CERR_NOT_SUPPORTED	Returned when abort is requested on TCP server or UDP channels
WIP_CERR_INTERNAL	Impossible to abort the TCP communication due to internal reasons

6.5.7. The wip_getOpts Function

The options supported by the wip_getOpts function, applied to a TCPClient are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_KEEPALIVE	u32* n	Sends a NOOP command every n tenth of seconds, so that the server and any NAT on the way won't shut down the connection default:1
WIP_COPT_SND_BUFSIZE	u32*	Size of the emission buffer associated with a socket. default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32*	Size of the reception buffer associated with a socket. default: depends on the protocol.
WIP_COPT_SND_LOWAT	u32*	Minimum amount of available space that must be available in the emission buffer before triggering a WIP_CEV_WRITE event. default:1024
WIP_COPT_RCV_LOWAT	u32*	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ event. default:1
WIP_COPT_ERROR	s32*	Number of the last error experienced by that socket. default:WM_EOK
WIP_COPT_NREAD	u32*	Number of bytes that can currently be read on that socket. default:0
WIP_COPT_NWRITE	u32*	Number of bytes that can currently be written on that socket. For a PING, size of the request default:20
WIP_COPT_NODELAY	bool*	When set to TRUE, TCP packets are sent immediately, even if the buffer is not full enough When set to FALSE, the packets will be sent either, a) by combining several small packets into a bigger packet b) when the data is ready to send and the stack is idle Note: Data has to be buffered and managed by the user application. There is no provision in Internet Library APIs to wait for data block to be fully filled before sending it default:FALSE
WIP_COPT_MAXSEG	u32*	Maximum size of TCP packets
WIP_COPT_TOS	u8*	Type of Service (cf. RFC 791) default:0
WIP_COPT_TTL	u8*	Time-To-Live for packets. default:64
WIP_COPT_PORT	u16*	Port occupied by this socket. default:0
WIP_COPT_STRADDR	ascii* buff, u32 buf_len	Local address of the socket. default:0
WIP_COPT_ADDR	wip_in_addr_t*	Local address of the socket, as a 32 bits integer. default:0

Option	Value	Description
WIP_COPT_PEER_PORT	u16*	Port of the peer socket. default:0
WIP_COPT_PEER_STRADDR	ascii* buff, u32 buf_len	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection. default:0
WIP_COPT_PEER_ADDR	wip_in_addr_t*	Address of the peer socket, as a 32 bits integer. default:0
WIP_COPT_SUPPORT_READ	none	Fails if the channel does not support wip_read() operations. If supported, does nothing.
WIP_COPT_SUPPORT_WRITE	none	Fails if the channel does not support wip_write() operations. If supported, does nothing.
WIP_COPT_REXMT_MAX	u32*	Gets the maximum time between TCP retransmissions. default:64 seconds
WIP_COPT_REXMT_MAXCNT	u32*	Gets the maximum number of retransmissions. default:12
WIP_COPT_NOTIMEWAIT	u32*	If set, the time-wait state that may occur after closing the socket is disabled
WIP_COPT_KEEP_INIT	u32*	Value of connection establishment timer. The default value is 75 seconds.
WIP_COPT_KEEP_IDLE	u32*	Value of the keepalive idle time, the default value is 7200 seconds (2 hours).
WIP_COPT_KEEP_INTVL	u32*	Interval time between keepalive probes, the default value is 50 seconds.

6.5.8. The wip_setOpts Function

The options supported by the wip_setOpts function, applied to TCP clients are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_KEEPALIVE	u32	Sends a NOOP command every n tenth of seconds, so that the server and any NAT on the way won't shut down the connection default:1
WIP_COPT_SND_BUFSIZE	u32	Size of the emission buffer associated with a socket. default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32	Size of the reception buffer associated with a socket. default: depends on the protocol.
WIP_COPT_SND_LOWAT	u32	Minimum amount of available space that must be available in the emission buffer before triggering a WIP_CEV_WRITE event. default:1024
WIP_COPT_RCV_LOWAT	u32	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ event. default:1
WIP_COPT_NODELAY	bool	When set to TRUE, TCP packets are sent immediately, even if the buffer is not full enough When set to FALSE, the packets will be sent either, a) by combining several small packets into a bigger packet b) when the data is ready to send and the stack is idle Note: Data has to be buffered and managed by the user application. There is no provision in Internet Library APIs to wait for data block to be fully filled before sending it default:FALSE
WIP_COPT_TOS	u8	Type of Service (cf. RFC 791) default:0
WIP_COPT_TTL	u8	Time-To-Live for packets. default:64
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.
WIP_COPT_REXMT_MAX	u32	Sets the maximum time between TCP retransmissions. default:64 seconds
WIP_COPT_REXMT_MAXCNT	u32	Sets the maximum number of retransmissions. default:12
WIP_COPT_NOTIMEWAIT	u32	If set, the time-wait state that may occur after closing the socket is disabled
WIP_COPT_KEEP_INIT	u32	Value of connection establishment timer. The default value is 75 seconds.
WIP_COPT_KEEP_IDLE	u32	Value of the keepalive idle time, the default value is 7200 seconds (2 hours).
WIP_COPT_KEEP_INTVL	u32	Interval time between keepalive probes, the default value is 50 seconds.

6.5.9. The wip_readOpts Function

The options supported by the wip_readOpts function, applied to a TCPClient are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_PEEK	bool (set)	When true, the message is not deleted from the buffer after reading, so that it can be read again. default:FALSE

6.5.10. The wip_writeOpts Function

The option supported by the wip_writeOpts function, applied to a TCPClient is:

Option	Value	Description
WIP_COPT_END	none	End of the option

6.6. Ping: ICMP Echo Request Handler

The ping service is presented as a channel. It does not support read/write operations, the only thing it can do is receive and react to WIP_CEV_PING events.

Ping channels will generate WIP_CEV_PING events when receiving network responses. The ping channel has a reception timeout, set by WIP_COPT_RCV_TIMEOUT. If a network response arrives before [timeout], a WIP_CEV_PING event is generated, with its [timeout] flag set to false. If the ping packet has been sent, but the response didn't arrive within [timeout], a WIP_CEV_PING is generated, but its [timeout] flag is set to TRUE. However, if the ping packet couldn't be emitted at all (invalid hostname, non-routable address, network down...), no WIP_CEV_PING is generated; only a WIP_CEV_ERROR describing why the packet couldn't be sent is emitted.

6.6.1. The wip_pingCreate Function

The wip_pingCreate function creates a channel supporting a ping session.

Prototype

```
wip_channel_t wip_pingCreate ( const ascii   *peerAddr,
                               wip_eventHandler_f   evHandler,
                               void   *ctx );
```

Parameters

peerAddr:

Address of host that the user wants to ping. This can be either a DNS address, or a numeric one in the form “xxx.xxx.xxx.xxx”.

evHandler:

The call back handler which receives the network events related to the socket. Possible event kinds are WIP_CEV_PING and WIP_CEV_ERROR.

ctx:

It is the user data to be passed to the event handler every time it is called.

Returned Values

This function returns

- the created channel
- NULL on error

6.6.2. The wip_pingCreateOpts Function

The wip_pingCreateOpts function creates a channel supporting a ping session. When a response arrives, a PING event is sent to the event handler. The response contains:

- a packet index from 0 to n-1, n being the number of sent packet sets with WIP_COPT_REPEAT
- a response time in milliseconds
- a Boolean indicating whether the packet arrived too late (after the timeout limit set by WIP_COPT_RCV_TIMEOUT)

Prototype

```
wip_channel_t wip_pingCreateOpts ( const ascii   *destAddr,
                                   wip_eventHandler_f   handler,
                                   void   *ctx,
                                   ... );
```

Parameters

destAddr:

Address of host that the user wants to ping. This can be either a DNS address, or a numeric one in the form “xxx.xxx.xxx.xxx”.

handler:

The call back handler which receives the network events related to the socket. Possible events kinds are WIP_CEV_PING and WIP_CEV_ERROR.

ctx:

It is the user data to be passed to the event handler every time it is called

... :

The parameters are the same as the parameters for the wip_pingCreate() function, plus a WIP_COPT_END-terminated series of option parameters. The options supported by wip_pingCreateOpts() are:

Option	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_REPEAT	s32	Number of PING echo requests to send. default:1
WIP_COPT_INTERVAL	u32	Time between two PING echo requests, in ms. default:1000
WIP_COPT_RCV_TIMEOUT	u32	For PING channels, timeout for ECHO requests. default:1000
WIP_COPT_TTL	u8	Time-To-Live for packets. default:64

Option	Value	Description
WIP_COPT_NWRITE	u32	Number of bytes that can currently be written on that socket. For a PING, size of the request default:20
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

Returned Values

This function returns

- the created channel on success
- NULL on error

6.6.3. The wip_getOpts Function

The options supported by the wip_getOpts function, applied to a ping are:

Options	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_REPEAT	s32*	Number of PING echo requests to send. default:1
WIP_COPT_INTERVAL	u32*	Time between two PING echo requests, in ms. default:1000
WIP_COPT_RCV_TIMEOUT	u32*	For PING channels, timeout for ECHO requests. default:1000
WIP_COPT_TTL	u8*	Time-To-Live for packets. default:64
WIP_COPT_NWRITE	u32*	Number of bytes that can currently be written on that socket. For a PING, size of the request default:20

6.6.4. The wip_setOpts Function

The options supported by the wip_setOpts function, applied to a ping are:

Options	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_INTERVAL	u32	Time between two PING echo requests, in ms. default:1000
WIP_COPT_RCV_TIMEOUT	u32	For PING channels, timeout for ECHO requests. default:1000
WIP_COPT_REPEAT	s32	Number of PING echo requests to send. default:1
WIP_COPT_TTL	u8	Time-To-Live for packets. default:64
WIP_COPT_NWRITE	u32	Number of bytes that can currently be written on that socket. For a PING, size of the request default:20
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

6.7. IP TUN/TAP Channel

The IP TUN/TAP channel is used to redirect IP traffic to the application. The application can also send IP datagrams into the IP routing system. The creation of the channel will create an IP TUN/TAP interface with the given local and destination addresses. By setting the routing table and enabling IP forwarding, the application can redirect part of the IP traffic to this interface. IP packets sent to the interface are received by the channel and packets written to the channel are received by the interface. Additional routes may be added in the routing table to send raw IP packets to the application in case destination IP addresses would match the IP TUN/TAP subnet but not its specific address. This way, once the data packets are received at the IP TUN/TAP interface level, they are automatically forwarded to the application.

6.7.1. Architecture of IP TUN/TAP Interface

A new type of channel, IP TUN/TAP, is defined to allow redirection of IP traffic to and from the application. The solution is similar to the IP TUN/TAP interface device (tun/tap) of Linux operating system. The IP TUN/TAP channel allows application to receive and send raw IP packets. Please refer to the section 4.14 for the details about IP routing management and section 3.5.2 for enabling the option WIP_NET_OPT_IP_FORWARD for forwarding of IP datagrams.

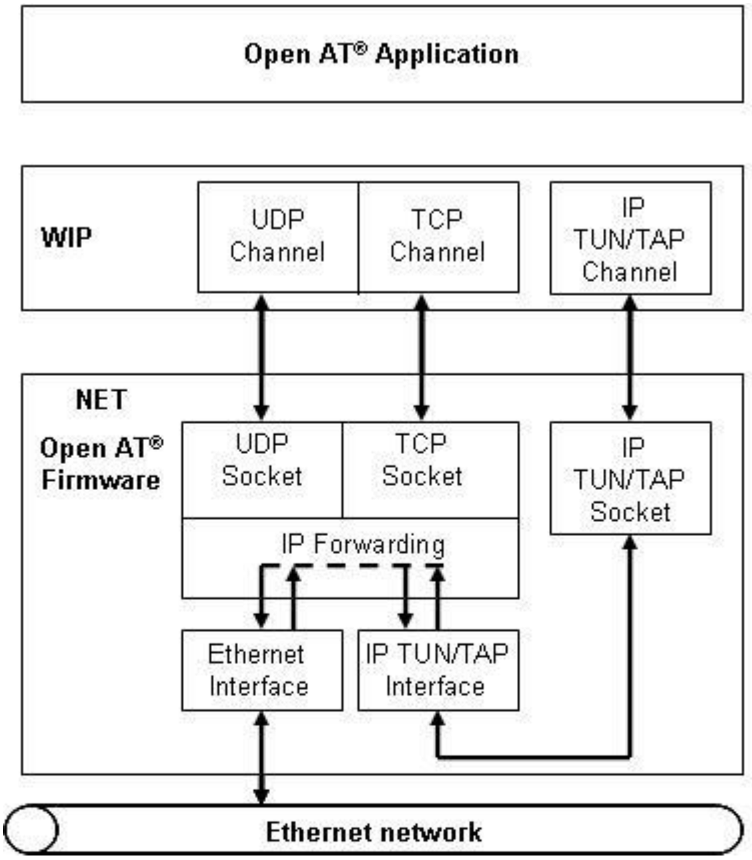


Figure 18. Architecture of IP TUN/TAP Interface

6.7.2. The wip_TUNCreate Function

The wip_TUNCreate function creates an IP TUN/TAP channel between the application and the TCP/IP stack with default options. At least either local or destination address must be specified in order to allow routing of datagrams to the associated network interface.

Note: By default, the number of IP tunnel interfaces available is set to 0 via the AT+WIPS command. The number of IP tunnel interfaces available for the application have to be specified using the AT+WIPS command prior to using the TUN/TAP APIs.

Prototype

```
wip_channel_t wip_TUNCreate( wip_in_addr_t   local_ip,

                             wip_in_addr_t   dest_ip,

                             wip_eventHandler_f   evHandler,

                             void   *ctx );
```

Parameters

local_ip:

In: Local or source IP address.

dest_ip:

In: Destination IP address.

evHandler:

In: Callback handler which receives the network events related to the channel created. WIP_CEV_OPEN, WIP_CEV_READ, WIP_CEV_WRITE and WIP_CEV_ERROR are the possible events that will be received in the event handler.

ctx:

In: User data to be passed to the event handler every time the function is called.

Returned Values

This function returns

- The created channel
- NULL on error

6.7.3. The wip_TUNCreateOpts function

The wip_TUNCreateOpts function creates an IP TUN/TAP channel between the application and the TCP/IP stack with advanced options. At least either local or destination address must be specified in order to allow routing of datagrams to the associated network interface.

Prototype

```
wip_channel_t wip_TUNCreateOpts( wip_in_addr_t   local_ip,
                                 wip_in_addr_t   dest_ip,
                                 wip_eventHandler_f   evHandler,
                                 void   *ctx,
                                 … );
```

Parameters

local_ip:

In: Local or source IP address.

dest_ip:

In: Destination IP address.

evHandler:

In: Callback handler which receives the network events related to the channel created. WIP_CEV_OPEN, WIP_CEV_READ, WIP_CEV_WRITE and WIP_CEV_ERROR are the possible events that will be received in the event handler.

ctx:

In: User data to be passed to the event handler every time the function is called.

...:

In: The option that is supported is mentioned in the table below.

Option	Value	Description
WIP_COPT_END	none	This option defines the end of the option list.

Returned Values

This function returns

- The created channel
- NULL on error

Note: The option WIP_NET_OPT_IP_FORWARD needs to be set to TRUE at the time of Internet Library stack initialization or before the IP TUN/TAP channel is created using either wip_netSetOpts or wip_netInitOpts function.

7. FILE

As in WIP, communication happens through abstract channels, called wip_channel_t. The control of a file resource such as FTP or HTTP will be ensured by a connection channel; variables holding a connection channel will typically be called cx. Whenever a connection channel has to transfer data, it will do so asynchronously, by creating a dedicated data transfer channel; variables holding data transfer channels will typically be called c.

For instance, when we want to send data to a connection channel, we will call wip_putFile(), which will return a data transfer channel. This channel will receive events related to the file transfer:

- WIP_CEV_OPEN when it is ready to receive data
- WIP_CEV_WRITE when data can be sent, and, if it went through an overflow of data to send, then becomes available again to send more data
- WIP_CEV_ERROR in case of underlying protocol error

Created data channel will also support wip_write(), so that the application can actually send the data which represent the file contents inside the WIP_CEV_WRITE event; finally, wip_close() will free the data transfer channel, and signal that the whole file has been written. wip_setOpts() allows to pass protocol-dependent settings to the channel.

Similarly, wip_getFile() will retrieve files from the connection, also by spawning a data transfer channel; this data transfer channel will experience WIP_CEV_OPEN, WIP_CEV_READ, WIP_CEV_ERROR events, and WIP_CEV_PEER_CLOSE once the whole file has been read. It also supports wip_read() and wip_close()."

When the data transfer is terminated, an event WIP_CEV_DONE is also sent to the connection channel used to start the file operation, so a new transfer can be started using the same connection channel.

File listing also implies asynchronous data transfer, and will also happen through a spawned data transfer channel, as detailed below.

Both connection channels and data transfer channels are supported by the same wip_channel_t C type. Indeed, connection and data transfer channels both support wip_setOpts(), wip_getOpts() and wip_close() functions (plus a couple of other, less important, functions), they must therefore share the same type. Moreover, some dynamic type checking is performed, so that if an application tries to use wip_getFile() on a data channel, or wip_read() on a connection channel, an explicit error message will be issued.

7.1. Required Header File

The header file for the FILE service is wip_file.h.

7.2. The wip_getFile Function

The wip_getFile function is used to download a file from the server. The connection channel is not used for reading a file content. Instead, this function create and return dedicated data transfer channel, which support read events and function calls.

7.2.1. Prototype

```
wip_channel_t wip_getFile ( wip_channel_t   ftp_cx,
                            ascii   *file_name,
                            wip_eventHandler_f   evh,
                            void   *ctx );
```

7.2.2. Parameters

ftp_cx:

It is the connection channel

file_name:

It is the name of the file to download from the server. Some protocols might support unnamed files; in this case, NULL is an acceptable value.

evh:

It is the event handler to be attached to the newly created data transfer channel. It is the responsibility of the event handler, provided by the user, to read the arriving data, and to put them in the appropriate place. When the file transfer is finished, a WIP_CEV_PEER_CLOSE event is sent to the event handler.

ctx:

It is the user data passed to the event handler, evh every time it is called.

7.2.3. Returned Values

The function returns

- data transfer channel on success
- NULL on failure

7.3. The wip_getFileOpts Function

The wip_getFileOpts function is used to download a file from the server with the user defined options like logging in with an account and password rather than anonymously. The connection channel is not used for reading a file content. Instead, this function creates and returns dedicated data transfer channel, which support read events and function calls.

7.3.1. Prototype

```
wip_channel_t wip_getFileOpts ( wip_channel_t   ftp_cx,
                                ascii   *file_name,
                                wip_eventHandler_f   evh,
                                void   *ctx,
                                ... );
```

7.3.2. Parameters

The parameters are the same as the parameters for the wip_getFile function, plus list of option names. The option names must be followed by option values. The list must be terminated by WIP_COPT_END. Supported options depend on the kind of connection channel and are mentioned in sections 8.8, 9.8 and 11.4.2.

7.3.3. Returned Values

The function returns

- data transfer channel on success
- NULL on failure

7.4. The wip_putFile Function

The wip_putFile function is used to upload a file to the server. The connection channel is not used for writing file content. Instead, these functions create and return dedicated data transfer channel, which supports write events and function calls.

7.4.1. Prototype

```
wip_channel_t wip_putFile ( wip_channel_t   ftp_cx,
                            ascii   *file_name,
                            wip_eventHandler_f   evh,
                            void   *ctx );
```

7.4.2. Parameters

ftp_cx:

It is the connection channel.

file_name:

It is the name of the file to upload on the server. Some protocols might support unnamed files; in this case, NULL is an acceptable value.

evh:

It is the event handler to be attached to the newly created data transfer channel. The possible event kind is WIP_CEV_WRITE.

ctx:

It is the user data passed to the event handler evh every time it is called.

7.4.3. Returned Values

The function returns

- data transfer channel on success
- NULL on failure

7.5. The wip_putFileOpts Function

The wip_putFileOpts function is used to upload a file to the server with the user defined options. The connection channel is not used for writing file content. Instead, these functions create and return dedicated data transfer channel, which supports write events and function calls.

7.5.1. Prototype

```
wip_channel_t wip_putFileOpts ( wip_channel_t   ftp_cx,

                                ascii   *file_name,

                                wip_eventHandler_f   evh,

                                void   *ctx,

                                ... );
```

7.5.2. Parameters

The parameters are the same as the parameters for the wip_putFile function, plus list of option names. The option names must be followed by option values. The list must be terminated by WIP_COPT_END. Supported options depend on the kind of connection channel and are mentioned in sections 8.10, 9.10 and 10.3.1.

7.5.3. Returned Values

The function returns

- data transfer channel on success
- NULL on failure

7.6. The wip_cwd Function

The wip_cwd function changes the current working directory on the server. Once this command is successfully terminated, a WIP_CEV_DONE event is sent to the event handler. If the change does not succeed (typically because dir_name doesn't exist in the current directory), a WIP_CEV_ERROR is sent to the handler.

The cx will be put in WIP_CSTATE_BUSY mode until the server response arrives, which means that no other command will be accepted by cx until WIP_CEV_DONE or WIP_CEV_ERROR arrives.

7.6.1. Prototype

```
int wip_cwd ( wip_channel_t   cx,

              ascii   *name );
```

7.6.2. Parameters

cx:

This is the connection channel whose working directory is to be changed.

name:

This is the name of the new working directory.

7.6.3. Returned Values

The function returns

- a status code 0 if the request has been sent successfully
- a negative error code on error

7.7. The wip_mkdir Function

The wip_mkdir function is used to create a new directory in the current working directory. The success or failure is reported as WIP_CEV_DONE or WIP_CEV_ERROR events on cx's event handler.

The cx will be put in WIP_CSTATE_BUSY mode until the server response arrives, which means that no other command will be accepted by cx until WIP_CEV_DONE or WIP_CEV_ERROR arrives.

7.7.1. Prototype

```
int wip_mkdir ( wip_channel_t   cx,

                ascii   *name );
```

7.7.2. Parameters

cx:

This is the connection channel whose working directory is to be changed.

name:

This is the name of the new working directory.

7.7.3. Returned Values

The function returns

- 0 on success
- negative error code on error

7.8. The wip_deleteFile Function

The wip_deleteFile function is used to delete a file. The success or failure is reported as WIP_CEV_DONE or WIP_CEV_ERROR events on cx's event handler.

The cx will be put in WIP_CSTATE_BUSY mode until the server response arrives, which means that no other command will be accepted by cx until WIP_CEV_DONE or WIP_CEV_ERROR arrives.

7.8.1. Prototype

```
int wip_deleteFile ( wip_channel_t   cx,

                     ascii   *name );
```

7.8.2. Parameters

cx:

This is the connection channel on which file will be deleted.

name:

It is the name of the file to delete.

7.8.3. Returned Values

The function returns

- 0 on success
- negative error code on error

Note: *The wip_deleteFile function returns "OK" as response irrespective of the status of deletion. The success or failure of the deletion operation is decided based on the WIP_CEV_DONE or WIP_CEV_ERROR event that will be received in the event handler. Negative error code is returned only if there is sockert error.*

7.9. The wip_deleteDir Function

The wip_deleteDir function is used to delete an empty directory. The success or failure is reported as WIP_CEV_DONE or WIP_CEV_ERROR events on cx's event handler.

The cx will be put in WIP_CSTATE_BUSY mode until the server response arrives, which means that no other command will be accepted by cx until WIP_CEV_DONE or WIP_CEV_ERROR arrives.

7.9.1. Prototype

```
int wip_deleteDir ( wip_channel_t   ftp_cx,

                    ascii   *dir_name );
```

7.9.2. Parameters

cx:

This is the Connection channel on which file will be deleted.

name:

This is the name of the directory to be deleted.

7.9.3. Returned Values

The function returns

- 0 on success
- negative error code on error

7.10. The wip_renameFile Function

The wip_renameFile function is used to change file name. The file is expected to be in the current working directory. The success or failure is reported as WIP_CEV_DONE or WIP_CEV_ERROR events on cx's event handler.

The cx will be put in WIP_CSTATE_BUSY mode until the server response arrives, which means that no other command will be accepted by cx until WIP_CEV_DONE or WIP_CEV_ERROR arrives.

7.10.1. Prototype

```
int wip_renameFile ( wip_channel_t   cx,
                     ascii   *old_name,
                     ascii   *new_name );
```

7.10.2. Parameters

cx:

This is the connection channel on which file will be renamed old_name.

old_name:

This is the previous name of the file.

new_name:

This is the new name to give to the file.

7.10.3. Returned Values

The function returns

- 0 on success
- negative error code on error

7.11. The wip_getFileSize Function

The wip_getFileSize function is used to get the file size in bytes. On success, a WIP_CEV_DONE event is sent to ftp_ctx, with
event->content.done.aux set to the file's size. On failure, a WIP_CEV_ERROR event is triggered.

7.11.1. Prototype

```
int wip_getFileSize ( wip_channel_t   cx,

                      ascii   *name );
```

7.11.2. Parameters

cx:

This is the connection channel of the file whose size is required.

name:

This is the name of the file whose size is required.

7.11.3. Returned Values

The function returns

- 0 on success
- negative error code on error

7.12. The wip_list Function

As for other kinds of data transfer with the network, directory listing must happen asynchronously. When the server replies, its reply is handled in the standard Internet Library way: a data transfer channel is created by the connection channel; information about files is gathered through wip_read, and the application is informed that data is available through WIP_CEV_READ events, preceded by an initial WIP_CEV_OPEN when the channel initialization is done.

Information arrives on the spawned data transfer channel in the form of wip_fileInfo_t structures:

```
typedef struct wip_fileInfo_t {
  u16 size;
  u16 nentries;
  union wip_fileInfo_entry_t {
  u32 _u32; s32 _s32;
  ascii *_ascii;
    void *ptr;
  } entries[1];
} wip_fileInfo_t;
```

This structure contains a table of data entries, which can be access through known index. For instance, FTP will define the following entry numbers:

```
enum {
  WIP_FOPT_NAME;
  WIP_FOPT_SIZE;
  WIP_FOPT_CANREAD;
  WIP_FOPT_CANWRITE;
  WIP_FOPT_ISDIRECTORY;
};
```

Values can be accessed by using these indexes on the entries. For instance, the following code displays the name and size of the file described by the wip_fileInfo_t structure:

```
ascii response [100]
wm_sprintf (response ,"The file %s is %i bytes long.\n",
                   fi.entries [WIP_FOPT_NAME].u32,
                   fi.entries [WIP_FOPT_SIZE].ascii);
adl_atSendResponse( ADL_AT_PORT_TYPE( adl_port, ADL_AT_RSP),
                                  response);
```

Event generation: The resulting channel from after wip_list function call is a stream channel i.e.

- a WIP_CEV_OPEN event is sent before the listing is ready to begin
- a WIP_CEV_READ is sent when the first chunk of data is available
- after a call to wip_read() failed to entirely fill the buffer, the next arrival of data is signaled by a new WIP_CEV_READ event

- a WIP_CEV_PEER_CLOSE after the last data is arrived

Reading on the channel: The channel is filled with wip_fileInfo_t structures. wip_read() will only write entire structures, therefore if the buffer size is not a multiple of sizeof(wip_fileInfo_t), it cannot be entirely filled. All file Info structures have been read when WIP_CEV_PEER_CLOSE event is received.

Structure initialization: Initializing a wip_fileInfo_t structure is quite difficult, due to various pointer settings and memory manipulations. A function wip_fileInfoInit() is provided to ease this.

7.12.1. Prototype

```
wip_channel_t wip_list ( wip_channel_t   cx,
                         ascii   *dir_name,
                         wip_eventHandler_f   evh,
                         void   *ctx );
```

7.12.2. Parameters

cx:

This is the Connection channel

dir_name:

This is the name of the directory whose content must be listed (can be NULL, in this case the CWD will be listed)

evh:

This is the Event handler which will receive the events

ctx:

This is the evh user data

7.12.3. Returned Values

The function returns spawned transfer channel.

7.13. The wip_fileInfoInit Function

Initializing a wip_fileInfo_t structure is quite difficult, due to various pointer settings and memory manipulations. A function wip_fileInfoInit() is provided to ease this.

7.13.1. Prototype

```
wip_fileInfo_t *wip_fileInfoInit ( void   *buffer,
                                   u32    buf_len,
                                   ...);
```

7.13.2. Parameters

buffer:

The memory area where the file Info structure will be built

buf_len:

The amount of memory available in buffer

...:

A list of entry descriptions, terminated with WIP_FOPT_END. Each description has one of the following forms:

- option index, WIP_FOPT_TYPE_U32
- option index, WIP_FOPT_TYPE_S32
- option index, WIP_FOPT_TYPE_PTR, data_len
- option index, WIP_FOPT_TYPE_ASCII, string_len

option_index will typically be a WIP_FOPT_XXX index.

If the **WIP_FOPT_TYPE** given is u32 or s32, then the integer entry is initialized to zero. If it is a **ptr** or an **ascii***, it is initialized as a pointer, in an area in the buffer after the **wip_fileInfo_t**, to a reserved memory area of **data_len** (resp. **string_len**) bytes. This area is initialized with zeros as well.

The field size and **nentries** of the returned **wip_fileInfo_t** structure are set to the correct values as well. **size** takes the additional memory areas (for ascii and ptr entries) into account.

Notice that the **WIP_FOPT_XXX** indexes do not need to be passed in increasing order, and do not need to be contiguous either. Any “gap” in the entries would be set to zero.

7.13.3. Returned Values

The function returns

- A pointer to the created wip_fileInfo_t structure on success; this pointer will be equal to buffer.
- NULL on error (most likely a “not enough memory” error)

8. FTP Client

FTP client offers the ability to transfer files to and from an FTP server, through TCP/IP. Sierra Wireless's FTP client has the following specificities:

- it is based on Sierra Wireless's wip_channel_t abstract channel interface, and its file transfer abstract API
- it does not rely on a local file system

An FTP session mainly consists of connection to the FTP server; this connection is represented as a wip_channel_t. This connection will support various operations, among which the most important are file getting and file putting. Whenever the user requires the FTP session to get or put a file from/to the server, a new data transfer connection is opened, which is intended to read/write the file from/to the server. Several FTP sessions can happen simultaneously, which means that the application can read/write several files concurrently. As FTP protocol does not support multiple file transfer over a single session, only one transfer at a time can happen on a FTP session.

8.1. Required Header File

The header file for the FTP service is wip_ftp.h.

8.2. The wip_FTPCreate Function

An anonymous FTP connection is created through a call to wip_FTPCreate. The wip_FTPCreate function takes an event handler as a parameter, which will be in charge of reacting to network-caused events on the FTP session.

The FTP connection is not ready as soon as the creation function returns. The user is notified that the connection is ready when WIP_CEV_OPEN event is received in the event handler. If the initialization fails (e.g., the password is not accepted, or the server is not reachable), a WIP_CEV_ERROR will be received in the event handler.

8.2.1. Prototype

```
wip_channel_t wip_FTPCreate ( ascii   *server_name,

                              wip_eventHandler_f   evh,

                              void   *ctx );
```

8.2.2. Parameters

server_name:

In: The name of the server, either as a DNS resolved name, or in dotted notation, e.g. “192.168.1.1”.

evh:

In: The event handler is the one that receives reactions from the network.

ctx:

In: This is the user data to be passed to the event handler every time it is called.

8.2.3. Returned Values

The function returns

- the created channel on success
- NULL on error

8.3. The wip_FTPCreateOpts Function

The wip_FTPCreateOpts function is used to create FTP connection with user defined options like, logging in with an account and password rather than anonymously.

8.3.1. Prototype

```
wip_channel_t wip_FTPCreateOpts ( ascii   *server_name,
                                  wip_eventHandler_f   evh,
                                  void   *ctx,
                                  ... );
```

8.3.2. Parameters

The parameters are the same as the parameters for the wip_FTPCreate() function, plus list of option names. The option names must be followed by option values. The list must be terminated by WIP_COPT_END .The options supported by wip_FTPCreateOpts() are:

Option	Value	Description
WIP_COPT_TYPE	ascii	Translation of carriage returns. 'I' for image (no translation, the default) 'A' for ASCII 'E' for EBCDIC
WIP_COPT_PASSIVE	bool	Active or Passive Default is passive mode
WIP_COPT_USER	ascii*	User name Default is "anonymous"
WIP_COPT_PASSWORD	ascii*	Password Default is "wipftp@sierrawireless.com"
WIP_COPT_ACCOUNT	ascii*	Account Default is empty string
WIP_COPT_PEER_PORT	u16	Server FTP port default is 21
WIP_COPT_LIST_PLUGIN	wip_eventHandler_f	Library handling the results from the LIST FTP command (non-standard, server-dependent)
WIP_COPT_KEEPALIVE	u32	Sends a NOOP command every n tenth of seconds, so that the server and any NAT on the way won't shut down the connection default:1
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

Note: The following options are now available for working with this function as well: WIP_COPT_SND_BUFSIZE, WIP_COPT_RCV_BUFSIZE, WIP_COPT_TOS, WIP_COPT_TTL, WIP_COPT_MAXSEG, WIP_COPT_SND_LOWAT, WIP_COPT_RCV_LOWAT, *WIP_COPT_NODELAY and WIP_COPT_KEEPALIVE. Their defaults are the same as the TCP sockets, and these options can be reached through wip_{get,set}Opts.*

8.3.3. Returned Values

The function returns

- the created channel on success
- NULL on error

8.4. The wip_setOpts Function

The FTP session channel accepts all TCP client options, since an FTP connection is a TCP socket.

The options supported by wip_setOpts function, applied to FTP are:

Options	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_KEEPALIVE	u32	Sends a NOOP command every n tenth of seconds, so that the server and any NAT on the way won't shut down the connection default:1
WIP_COPT_SND_BUFSIZE	u32	Size of the emission buffer associated with a socket. default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32	Size of the reception buffer associated with a socket. default: depends on the protocol.
WIP_COPT_TYPE	ascii	Transition of carriage returns. “I” for image (no transition, the default) “A” for ASCII “E” for EBCDIC
WIP_COPT_PASSIVE	bool	Active or Passive Default is passive mode
WIP_COPT_LIST_PLUGIN	wip_eventHandler_f	Library handling the results from the LIST FTP command (non-standard, server-dependent)
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

Refer to The wip_setOpts Function section for more details on wip_setOpts function.

8.5. The wip_getOpts Function

The FTP session channel accepts all TCP client options, since an FTP connection is a TCP socket.

The options supported by wip_getOpts function, applied to FTP are:

Options	Value	Description
WIP_COPT_END	none	End of the option
WIP_COPT_SND_BUFSIZE	u32*	Size of the emission buffer associated with a socket. default: depends on the protocol.
WIP_COPT_RCV_BUFSIZE	u32*	Size of the reception buffer associated with a socket. default: depends on the protocol.
WIP_COPT_ERROR	s32*	Number of the last error experienced by that socket. default: WM_EOK
WIP_COPT_PORT	u16*	Port occupied by this socket. default:0
WIP_COPT_STRADDR	ascii* buff, u32 buf_len	Local address of the socket. default:0
WIP_COPT_ADDR	wip_in_addr_t*	Local address of the socket, as a 32 bits integer. default:0
WIP_COPT_PEER_PORT	u16*	Port of the peer socket. default:0
WIP_COPT_PEER_STRADDR	ascii* buff, u32 buf_len	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection. default:0
WIP_COPT_PEER_ADDR	wip_in_addr_t*	Address of the peer socket, as a 32 bits integer. default:0
WIP_COPT_SUPPORT_READ	none	Fails if the channel does not support wip_read() operations. If supported, does nothing.
WIP_COPT_SUPPORT_WRITE	none	Fails if the channel does not support wip_write() operations. If supported, does nothing.
WIP_COPT_TYPE	ascii	Transition of carriage returns. "I" for image (no transition, the default) "A" for ASCII "E" for EBCDIC
WIP_COPT_PASSIVE	bool	When set, TCP packets are sent immediately, even if the buffer is not full enough.
WIP_COPT_LIST_PLUGIN	wip_eventHandler_f	Library handling the results from the LIST FTP command (non-standard, server-dependent)

Refer to The wip_getOpts Function section for more details on wip_getOpts function.

8.6. The wip_close Function

The FTP session is closed with wip_close function. Refer to The wip_close Function section for more details on wip_close function.

8.7. The wip_getFile Function

The function wip_getFile is used to download a file from the FTP server. Refer to The wip_getFile Function section for more details on wip_getFile function.

8.8. The wip_getFileOpts Function

The wip_getFileOpts function is used to download a file from the FTP server with user defined options. The options supported by the wip_getFileOpts function, applied to a FTP are the same WIP_COPT_XXX options as TCP client channels, plus the options which are mentioned below:

Option	Value	Description
WIP_COPT_FILE_NAME	ascii*, u32	Name of the file being received
WIP_COPT_RESTART	u32 n	Restart the transfer at the nth byte
WIP_COPT_END	none	End of the option

Refer to The wip_getFileOpts Function section for more details on wip_getFileOpts function.

8.9. The wip_putFile Function

The wip_putFile function is used to upload a file to the FTP server. Refer to the the wip_putFile function section for more details on wip_putFile function.

8.10. The wip_putFileOpts Function

The wip_putFileOpts function is used to upload a file to the server with the user defined options. The options supported by the wip_putFileOpts function, applied to a FTP are the same WIP_COPT_XXX options as TCP client channels, plus the options which are mentioned below:

Option	Value	Description
WIP_COPT_FILE_NAME	ascii*, 32	Name of the file being received
WIP_COPT_APPEND	bool	Append data to the end of the file
WIP_COPT_RESTART	u32 n	Restart the transfer at the nth byte
WIP_COPT_END	none	End of the option

Refer The wip_putFileOpts Function section for more details on wip_putFileOpts function.

8.11. The wip_shutdown Function

The wip_shutdown function is used to signal the end of file. For more details on wip_shutdown function, refer to The wip_shutdown Function section.

9. HTTP Client

HTTP client provides an application interface for generating HTTP requests using Sierra Wireless TCP/IP implementation (Internet Library). It is based on Internet Library abstract channel interface. The following features are provided:

- support for HTTP version 1.1 (default) and 1.0
- persistent connections (with HTTP 1.1)
- connection to a HTTP proxy server
- basic and digest (MD5) authentication
- chunked transfer coding
- setting HTTP request headers
- getting HTTP response headers
- GET, HEADER, POST and PUT methods

HTTP requests are generated in two phases. First, application must create a HTTP session channel with wip_HTTPCreate() or wip_HTTPCreateOpts() that will store information common to all further HTTP requests like

- HTTP version
- address of proxy server
- HTTP request headers

This channel will also maintain persistent connections. A new request channel is then created for each HTTP request using wip_getFile()/wip_getFileOpts() or wip_putFile()/wip_putFileOpts().

9.1. Required Header File

The header file for the HTTP client interface definitions is wip_http.h.

9.2. The wip_httpVersion_e Type

The wip_httpVersion_e type defines the HTTP version of the session.

```
typedef enum {
WIP_HTTP_VERSION_1_0,  /* HTTP 1.0 */
WIP_HTTP_VERSION_1_1   /* HTTP 1.1 */
} wip_httpVersion_e;
```

Refer to the RFC 2616 for detailed description of different HTTP versions.

9.3. The wip_httpMethod_e Type

The wip_httpMethod_e type defines the HTTP method of a message.

```
typedef enum {
  WIP_HTTP_METHOD_GET,            /* HTTP GET method */
  WIP_HTTP_METHOD_HEAD,           /* HTTP HEAD method */
  WIP_HTTP_METHOD_POST,           /* HTTP POST method */
  WIP_HTTP_METHOD_PUT,            /* HTTP PUT method */
  WIP_HTTP_METHOD_DELETE,         /* HTTP DELETE method */
  WIP_HTTP_METHOD_TRACE,          /* HTTP TRACE method */
  WIP_HTTP_METHOD_CONNECT         /* HTTP CONNECT method */
} wip_httpMethod_e;
```

Refer to the RFC 2616 for detailed description of different HTTP methods.

9.4. The wip_httpHeader_t Structure

The wip_httpHeader_t structure defines a HTTP header field.

```
typedef struct {
  ascii *name;      /* field name*/
  ascii *value;     /* field value*/
} wip_httpHeader_t;
```

The field name contains the name of the header and is case insensitive.

The field value contains the value of the header.

The headers are transmitted in the form “<header>: <value><CR><LF>” in the HTTP request and response messages.

9.5. The wip_HTTPClientCreate Function

The wip_HTTPClientCreate function is used to create HTTP session channels

9.5.1. Prototype

```
wip_channel_t wip_HTTPClientCreate ( wip_eventHandler_f   handler,
                                     void   *ctx );
```

9.5.2. Parameters

`handler`:

The call back handler which receives the network events related to the channel. Currently no event is defined so it should be set to NULL.

`ctx`:

This is the user data to be passed to the event handler every time it is called.

9.5.3. Returned Values

The function returns

- the created session channel
- else NULL on error

9.6. The wip_HTTPClientCreateOpts Function

The wip_HTTPClientCreateOpts function is used to create HTTP session channels with user defined options.

9.6.1. Prototype

```
wip_channel_t wip_HTTPClientCreateOpts ( wip_eventHandler_f   handler,
                                         void   *ctx,
                                         ... );
```

9.6.2. Parameters

The parameters are the same as the parameters for the wip_HTTPClientCreate function, plus list of configuration option names. The option names must be followed by option values and the list must be terminated by WIP_COPT_END. Each option can be followed by one or more values.

Option	Value	Description
WIP_COPT_END	none	This option defines the end of the option list.
WIP_COPT_RCV_BUFSIZE	u32	This option sets the size of the TCP socket receive buffer; the default value is specified by the TCP channel. default: depends on the protocol.
WIP_COPT_SND_BUFSIZE	u32	This option sets the size of the TCP socket send buffer; the default value is specified by the TCP channel. default: depends on the protocol.
WIP_COPT_HTTP_PROXY_STRADDR	ascii *	This option sets the hostname of the HTTP proxy server; a NULL value disables the proxy server. default:0
WIP_COPT_HTTP_PROXY_PORT	u16	This option sets the port number of the HTTP proxy server, the default value is 80.
WIP_COPT_HTTP_VERSION	wip_httpVersion_e	This option defines the HTTP version to be used by the session;the default version is HTTP 1.1
WIP_COPT_HTTP_DATA_ENCOD	bool	This option defines if the data to transfer are encoded (CHUNKED DATA) or not; the default version is TRUE
WIP_COPT_HTTP_HEADER	ascii *, ascii *	This option adds a HTTP message header field that will be sent on each request. The first value is the field name (without the colon), the second value is the field value (without CRLF), and a NULL value can be passed to remove a previously defined header field. default:0

Option	Value	Description
WIP_COPT_HTTP_HEADER_LIST	wip_httpHeader_t *	This option adds a list of HTTP message header fields to send with each request. The value points to an array of wip_httpHeader_t structures, the last element of the array has its name field set to NULL.
WIP_COPT_USER	ascii *	This option sets the user's name used for basic authentication.
WIP_COPT_PASSWORD	ascii *	This option sets the password used for basic authentication.
WIP_COPT_HTTP_MAXREDIRECT	u32	This option sets the maximum number of redirects that are allowed. Automatic redirect is only supported with GET method. A zero value disables automatic redirects; the default value is 8 redirects.
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

9.7. The wip_getFile Function

The wip_getFile function is used to send a HTTP request to the given URL. By default a HTTP GET request is sent, but other HTTP methods can be sent by this function using WIP_COPT_HTTP_METHOD option.

The URL used must be a valid HTTP URL as specified by RFC 2616; however some of its fields are optional like:

- if the protocol is not specified, “http:” protocol is assumed
- if the host name is not specified, the host name and port number of the previous request is used
- if a username and password are specified, they are used for authentication

When HTTP 1.1 is used, a new TCP channel is not created for each request destined to the same server or proxy; instead the TCP connection is maintained by the HTTP session whenever possible. If a different server is specified or if HTTP 1.0 protocol is specified, a new TCP channel is created for the request.

The events which are received in the event handler are listed below.

Event	Description
WIP_CEV_OPEN	This event is sent when the response message header has been received.
WIP_CEV_READ	This event is sent when response message body data is available for reading by the application.
WIP_CEV_PEER_CLOSE	This event is sent after the entire response message, including response header and response body data, has been received.
WIP_CEV_WRITE	This event is sent when request message body data can be written by the application.
WIP_CEV_ERROR	This event is sent when a socket error has occurred. The error number is passed in the content.error.errnum field of the event data structure.

When WIP_CEV_READ or WIP_CEV_PEER_CLOSE event has been received, the wip_getOpts function can be used to retrieve following response header information:

- WIP_COPT_HTTP_STATUS_CODE returns the 3-digit response status code,
- WIP_COPT_HTTP_STATUS_REASON returns the reason phrase,
- WIP_COPT_HTTP_HEADER returns the value of response header fields.

Note: *When the WIP_CEV_READ event is received and the data is not read, the subsequent WIP_CEV_PEER_CLOSE event is not received.*

Refer to The wip_getFile Function section for more details on wip_getFile function.

9.8. The wip_getFileOpts Function

The wip_getFileOpts function is used to send a HTTP request to the given URL with user defined options. The events which are received in the event handler are same as in the wip_getFile Function section.

The options supported by the wip_getFileOpts function, applied to a HTTP are:

Option	Value	Description
WIP_COPT_END	none	This option defines the end of the option list.
WIP_COPT_HTTP_METHOD	wip_httpMethod_e	This option defines the method of the HTTP message. The default method is WIP_HTTP_METHOD_GET; the other supported methods are WIP_HTTP_METHOD_HEAD, WIP_HTTP_METHOD_POST and WIP_HTTP_METHOD_PUT.
WIP_COPT_HTTP_HEADER	ascii *,ascii *	This option adds a HTTP message header field to the request. The first value is the field name (without the colon); the second value is the field value (without CRLF). This option overwrite fields previously defined by the session channel, a NULL value can be passed to remove a previously defined header field. default:0
WIP_COPT_HTTP_HEADER_LIST	wip_httpHeader_t *	This option adds a list of HTTP message header fields to the request. The value points to an array of wip_httpHeader_t structures, the last element of the array has its name field set to NULL.
WIP_COPT_HTTP_MAXREDIRECT	u32	This option sets the maximum number of redirects that are allowed. Automatic redirect is only supported with GET method. A zero value disables automatic redirects; the default value is 8 redirects. When specified, this option overwrites the value set on the session channel.
WIP_COPT_USER	ascii *	This option sets the user's name for basic authentication. When specified, this option overwrites the value set on the session channel, a NULL value disables authentification.
WIP_COPT_PASSWORD	ascii *	This option sets the password used for basic authentication. When specified, this option overwrites the value set on the session channel, a NULL value disables authentification.

Refer to the the wip_getFileOpts Function section for more details on wip_getFileOpts function.

9.9. The wip_putFile Function

The wip_putFile function sends a HTTP PUT request to the given URL. The events received in the event handler are same as defined for the wip_getFile function in the wip_getFile Function section..

For more details on wip_putFile function, refer the the wip_putFile Function section.

Note: The only difference with wip_getFile is the default HTTP method used.

9.10. The wip_putFileOpts Function

The wip_putFileOpts function sends a HTTP PUT request to the given URL with the user defined options. Refer to the the wip_putFileOpts Function section for more details on the syntax of the function.

Refer to the the wip_getFileOpts Function section for a description of supported options and the wip_getFile Function section for the events received in the event handler.

9.11. The wip_read Function

The wip_read function is used to read the response message body.

For more details on wip_read function, refer to the the wip_read Function section.

Note: This function is not supported by session channels.

9.12. The wip_write Function

The wip_write function is used to write the request message body. Not all requests have a message body.

For more details on wip_write function, refer to the the wip_write Function section.

Note: This function is not supported by session channels.

9.13. The wip_shutdown Function

The wip_shutdown function is used on a request channel to signal the end of the message body. This function has no effect if the request has no message body.

For more details on wip_shutdown function, refer to the the wip_shutdown Function section.

Note: This function is not supported by session channels.

The “read” parameter of this function is not used and must be set to FALSE.

The “write” parameter of this function must be set to TRUE to indicate that all the message body of the request has been transferred. If HTTP 1.1 is used; the TCP communication stays open for further requests.

9.14. The wip_setOpts Function

The wip_setOpts function is used to set or change options on a session channel, there is no option currently defined for a request channel.

Each option can be followed by one or more values. Refer wip_HTTPClientCreateOpts function for a description of supported options.

For more details on wip_setOpts function, refer to the the wip_setOpts Function section.

9.15. The wip_getOpts Function

The wip_getOpts function is used to retrieve options of a session or request channel.

Session channel supports the following options:

Option	Value	Description
WIP_COPT_END	none	This option defines the end of the option list.
WIP_COPT_RCV_BUFSIZE	u32*	This option returns the current size of the TCP socket receive buffer. default: depends on the protocol.
WIP_COPT_SND_BUFSIZE	u32*	This option returns the current size of the TCP socket send buffer. default: depends on the protocol.
WIP_COPT_HTTP_PROXY_STRADDR	ascii*	This option returns the hostname of the HTTP proxy server. default:0
WIP_COPT_HTTP_PROXY_PORT	u16*	This option returns the port number of the HTTP proxy server.
WIP_COPT_HTTP_VERSION	wip_httpVersion_e *	This option returns the selected HTTP version.

Request channel supports the following options:

Option	Value	Description
WIP_COPT_END	none	This option defines the end of the option list.
WIP_COPT_HTTP_STATUS_CODE	u32*	This option returns the 3-digit status code of the response.
WIP_COPT_HTTP_STATUS_REASON	ascii*, u32	This option returns the reason phrase of the response, the first value points to the buffer where the reason phrase is to be written, the second value is the size of the buffer.
WIP_COPT_HTTP_HEADER	ascii*, ascii*, u32	This option returns the value of the HTTP message header field with the name given by the first value, the second value points to the buffer where the field value is to be written, the third value is the size of the buffer. default:0

Refer to the the wip_getOpts Function section for more details on wip_getOpts function.

9.16. The wip_abort Function

The wip_abort function is only supported by the session channel. This call closes the current persistent connection, if any. If a request is pending the request is aborted.

For more details on wip_abort function, refer to the the wip_abort Function section.

9.17. The wip_close Function

On the session channel, the wip_close function aborts any current request and release resources associated with the session channel. This does not close the request channel.

On a request channel, the wip_close function closes the channel and makes the session ready for another request. When HTTP1.1 is used this does not close the TCP communication channel, it can be reused if the next request is sent to the same server. If the request is not completed when wip_close() is called, the TCP communication is reset to indicate to the server that the request was incomplete.

For more details on wip_close function, refer to the the wip_close Function section.

9.18. HTTP Authentication Helper Functions

The HTTP client provides embedded support for basic authentication. Other types of authentication may be used with HTTP, like digest authentication. These other schemes can be implemented at the application level by providing the authentication header(s) to the HTTP request.

9.18.1. Required Header File

The required header file for HTTP authentication helper functions is wip_http.h.

9.18.2. The wip_HTTPAuthScheme Function

The wip_HTTPAuthScheme function parses the “www-authenticate” authentication response header returned by the server to check if the given authentication scheme is supported.

```
bool wip_HTTPAuthScheme ( const ascii   *wwwauth,
                          const ascii   *scheme );
```

Parameters

`wwwauth:`

The value of the “www-authenticate” response header.

`Scheme:`

The name of the authentication scheme to test (case insensitive), for example “basic” or “digest”.

Returned Values

The function returns

- TRUE if the authentication scheme is present in the response header
- FALSE if the authentication scheme is not present in the response header

9.18.3. The wip_HTTPAuthBasic Function

The wip_HTTPAuthBasic function constructs an “authorization” request header using basic authentication scheme. The description of the basic authentication and the different parameters can be found in RFC 2617.

```
ascii *wip_HTTPAuthBasic ( const ascii   *user,

                           const ascii   *passwd);
```

Parameters

user:

The user’s name.

passwd:

The password.

Returned Values

The function returns a string that contains the value of “authorization” header to send in the request. The string is dynamically allocated and must be released with wip_HTTPAuthFree function.

9.18.4. The wip_HTTPAuthDigest Function

The wip_HTTPAuthDigest function constructs an “authorization” request header using digest authentication scheme. The description of the digest authentication and the different parameters can be found in RFC 2617.

```
ascii *wip_HTTPAuthDigest ( const ascii *wwwauth,
                            const ascii   *url,
                            wip_httpMethod_e   method,
                            const ascii   *cnonce,
                            u32   non_count,
                            const ascii  *user,
                            const ascii  *passwd);
```

Parameters

wwwauth:

The value of “www-authenticate” header returned by the server.

url:

The URL of the request.

method:

The HTTP method of the request.

cnonce:

The value of the “cnonce” field.

non_count:

The value of the “non-count” field.

user:

The user’s name.

passwd:

The password.

Returned Values

The function returns a string that contains value of the “authorization” header to send in the request. The string is dynamically allocated and must be released with wip_HTTPAuthFree function.

9.18.5. The wip_HTTPAuthFree Function

The wip_HTTPAuthFree function releases a buffer returned by the wip_HTTPAuthBasic or wip_HTTPAuthDigest function.

```
bool wip_HTTPAuthFree ( ascii   *auth );
```

Parameters

auth:

The string to release.

Returned Values

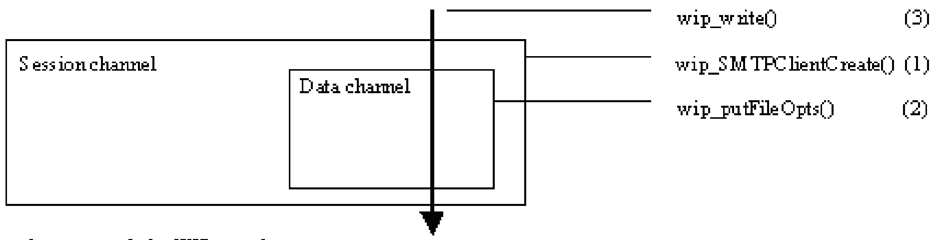
None

10. SMTP Client API

The SMTP (Simple Mail Transfer Protocol) is a standard protocol for mail transfer and delivery between Internet Hosts in a reliable and efficient manner. It requests using Sierra Wireless TCP/IP implementation (Internet Library). It is based on Internet Library abstract channel interface

SMTP mail sending process is generated in several phases:

- First, the application must create a SMTP session/connection channel with the interface wip_SMTPClientCreate() or wip_SMTPClientCreateOpts() that will store information common to all further SMTP requests: address of the mail server, authentication parameters. This channel will also maintain persistent connections.
- A DATA channel is then created for each SMTP request using wip_putFileOpts(): the created DATA channel will store the information as sender name, sender address, (main, cc and bcc) recipients lists, subject of the mail.
- The message body content is then sent over the DATA channel with the wip_write() interface.



Mail sending steps schematic with the WIP interface:

Figure 19. Mail Sending Steps

10.1. Required Header File

The header file for the SMTP client interface definitions is wip_smtp.h.

10.2. The Session / Connection Channel

10.2.1. The wip_SMTPClientCreate Function

The wip_SMTPClientCreate () function is used to create an SMTP SESSION channel.

Note: *The wip_SMTPClientCreate () is used to create a SMTP session to a server that does not require authentication.*

Prototype

```
wip_channel_t wip_SMTPClientCreate ( ascii   *server,
                                     wip_eventHandler_f   handler,
                                     void   *ctx );
```

Parameters

server:

The name of the server: either as a DNS resolved name, or in dotted notation, e.g. “192.168.1.1”.

handler:

The call back handler which receives the network events related to the channel.

The events defined in the table below are supported.

Event	Description
WIP_CEV_OPEN	This event is received when the session channel is established.
WIP_CEV_PEER_CLOSE	This event is received when the peer closes down.
WIP_CEV_ERROR	This event is received when a socket error or protocol error occurs during the session. An error code can either be positive or a negative value depending on the cause of error. The error code will be negative, if error is due to Internet Library library socket. The error code will be positive, if error is due to SMTP protocol or Internet Library SMTP library. The wip_getOpts function can be used to determine the cause of an error. Refer section 16.4 for error code management.
WIP_CEV_DONE	This event is received when the data channel is closed to end the mail sending and that the transaction has ended properly. Otherwise, WIP_CEV_ERROR event will be received.

ctx:

This is the user data to be passed to the event handler every time it is called.

Returned value

The function returns

- the created session channel,
- else NULL on error

10.2.2. The wip_SMTPClientCreateOpts Function

The wip_SMTPClientCreateOpts () can be used to create SMTP session channel with the additional configuration options.

Prototype

```
wip_channel_t wip_SMTPClientCreateOpts ( ascii   *server,
                                         wip_eventHandler_f   handler,
                                         void   *ctx,
                                         ... );
```

Parameters

The parameters are same as the parameters of the wip_SMTPClientCreate function, plus list of configuration option names. The option names must be followed by option values. The option list must be terminated by WIP_COPT_END and each option can be followed by one value. The supported options and their associated values are defined in the table below:

Option	Value	Description
WIP_COPT_END	none	This option defines the end of the option list.
WIP_COPT_PEER_PORT	u16	This option sets the port number of the SMTP mail server, the default value is 25.
WIP_COPT_USER	ascii *	This option sets the username; the default value is “NULL.” Limited to 64 characters
WIP_COPT_PASSWORD	ascii *	This option sets the password; the default value is “NULL”. Limited to 64 characters
WIP_COPT_SMTP_AUTH_TYPE	wip_smtpClientAuthTypes_e	This option sets authentication type used for authentication. WIP_SMTP_AUTH_NONE (default value) : no authentication required WIP_SMTP_AUTH_CLEAR: authentification with no encryption WIP_SMTP_AUTH_MIME64: authentication used with encrypted username/password in MIME64 during AUTH LOGIN phase.
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

Note: *The following options are now available for working with this function as well: WIP_COPT_SND_BUFSIZE, WIP_COPT_RCV_BUFSIZE, WIP_COPT_TOS, WIP_COPT_TTL, WIP_COPT_MAXSEG, WIP_COPT_SND_LOWAT, WIP_COPT_RCV_LOWAT, WIP_COPT_NODELAY and WIP_COPT_KEEPALIVE. Their defaults are the same as the TCP sockets, and these options can be reached through wip_{get,set}Opts.*

Returned value

The function returns

- the created SESSION channel
- else NULL on error

Note: For the username and password the memory allocations should be managed by the the application.

10.2.3. The wip_getOpts Function

The wip_getOpts() function is used to retrieve options of a SESSION channel. The options supported by the wip_getOpts() function, applied to SMTP session channel are:

Option	Value	Description
WIP_COPT_ERROR	u32*	This event is received when a socket error or protocol error occurs during the session. An error code can either be positive or a negative value depending on the cause of error. The error code will be negative, if error is due to Internet Library socket. The error code will be positive, if error is due to SMTP protocol or Internet Library SMTP library. The wip_getOpts function can be used to determine the cause of an error. default:WM_EOK Refer to the SMTP Error Codes section for error codes.
WIP_COPT_SMTP_STATUS_CODE	u32*,ascii **	Returns two arguments: 1) the last error code / protocol error code (status) 2) and the associated response string Refer to the SMTP Error Codes section for error codes.
WIP_COPT_ADDR	ascii **	Get the specified hostname default:0
WIP_COPT_USER	ascii **	Get the specified username
WIP_COPT_PASSWORD	ascii **	Get the specified password
WIP_COPT_PEER_PORT	u32*	Get the specified port default:0
WIP_COPT_SMTP_AUTH_TYPE	u32*	Get the specified authentication type wip_smtpClientAuthTypes_e

Refer section 6.2.6 for more details on the wip_getOpts function.

10.2.4. The wip_close Function

On the SESSION channel the wip_close() function aborts any current request and release resources associated with the session channel.

Note: This interface does not close the opened DATA channel. It is the application which is in charge of closing the opened channels

Refer section 6.2.1 for more details on wip_close function.

10.3. The Data Channel

10.3.1. The wip_putFileOpts Function

The wip_putFileOpts function is used to open DATA channel for the requested mail transfer by passing additional / mandatory configuration options. The events defined in the table below are supported.

Event	Comment
WIP_CEV_OPEN	This event is received when the DATA channel is established and ready for data sending
WIP_CEV_WRITE	This event is received when mail body data or mail body data with the header information can be written by an application.
WIP_CEV_ERROR	This event is received when a socket error or protocol error occurs during the session. An error code can either be positive or a negative value depending on the cause of error. The error code will be negative, if error is due to Internet Library library socket. The error code will be positive, if error is due to SMTP protocol or Internet Library SMTP library. The wip_getOpts function can be used to determine the cause of an error. Refer section16.4 for details on error codes.

The options supported by the wip_putFileOpts() function, applied to a SMTP Client are:

Option	Value	Comment
WIP_COPT_END	none	This option defines the end of the option list.
WIP_COPT_SMTP_SENDERNAME	ascii *	This option sets sender name default:0
WIP_COPT_SMTP_SENDER	ascii *	This option sets sender Email address (mandatory option) default:0
WIP_COPT_SMTP_REC	ascii *	This option sets recipients addresses list pointer (mandatory option) default:0
WIP_COPT_SMTP_CC_REC	ascii *	This option sets Carbon Copy Recipients addresses list pointer (mandatory option) default:0
WIP_COPT_SMTP_BCC_REC	ascii *	This option sets Blind Carbon Copy Recipients addresses list pointer (mandatory option) default:0
WIP_COPT_SMTP_SUBJ	ascii *	This option sets subject of the mail
WIP_COPT_SMTP_FORMAT_HEADER	u32	This option indicates whether header information should be generated by Internet Library library or by an application. 0: indicates that the header information will be generated and formatted by application before sending the mail content. This option is useful when application wants to format its header for example, while sending attached documents. 1(default case): indicates that the header information will be generated by Internet Library library.

Note: *The Email addresses for the field SENDER / RCPT / CC_RCPT and BCC_RCPT are provided in literal format, for instance : sender@domain.fr or <sender@domain.fr> depending on server implementation.*

The Email list for RCPT / CC_RCPT and BCC_RCPT fields, separator character is a coma "," for instance: rec01@domain.fr, rec02@domain.fr

An application is responsible for allocating memory for the options RCPT/CC_RCPT/BCC_RCPT list.

One of the recipient list and the sender address field are mandatory in order to be able to send a mail. When sender or recipients are not properly formatted, an WIP_CEV_ERROR event will be received.

Refer to section 7.5 for more details on wip_putFileOpts function.

10.3.2. The wip_getOpts Function

The wip_getOpts function is used to retrieve options of a session or request channel. The options supported by the wip_getOpts function, applied to a SMTP Client are:

Option	Value	Comment
WIP_COPT_ERROR	u32*	This event is received when a socket error or protocol error occurs during the session. An error code can either be positive or a negative value depending on the cause of error. The error code will be negative, if error is due to Internet Library socket. The error code will be positive, if error is due to SMTP protocol or Internet Library SMTP library. The wip_getOpts function can be used to determine the cause of an error. default:WM_EOK Refer to the SMTP Error Codes section for error codes.
WIP_COPT_SMTP_STATUS_CODE	u32,ascii**	Return 2 arguments: 1) the last error code / protocol error code (status) 2) and the associated response string Refer to the SMTP Error Codes section for error codes.
WIP_COPT_SMTP_SENDERNAME	ascii*	Get the sender name default:0
WIP_COPT_SMTP_SENDER	ascii*	Get the sender email address (mandatory option) default:0
WIP_COPT_SMTP_REC	ascii*	Get the recipients addresses list pointer (mandatory option) default:0
WIP_COPT_SMTP_CC_REC	ascii*	Get the Carbon Copy Recipients addresses list pointer (mandatory option) default:0
WIP_COPT_SMTP_BCC_REC	ascii*	Get Blind Carbon Copy Recipients addresses list pointer (mandatory option) default:0
WIP_COPT_SMTP_FORMAT_HEADER	u32*	Get the configured mode: 1: header will be generated by Internet Library 0: no header generated by the Internet Library default:1

Refer to section 6.2.6 for more details on the wip_getOpts function.

10.3.3. The wip_write Function

The wip_write function is used to write the request mail data through an opened data channel (previously opened with a wip_putfileOpts function).

Note: *The wip_write will transfer the mail data in plain text as formatted by the application without any encoding process. The application is responsible of choosing the appropriated encoding algorithm for the data to send.*

Moreover, if the 5 characters string <CR><LF>.<CR><LF> (hexdecimal: 0x0d 0x0a 0x2E 0x0d 0x0a) is present in the message body, the mail transfer will be completed and sent; therefore application should ensure that this 5 characters string is not present in the message body.

For encoding, the MIME specifications provides the standard mechanisms for structured message bodies

Refer to section 6.2.4 for more details on the wip_write function.

10.3.4. The wip_close Function

On a DATA channel the wip_close function closes the DATA channel and completed the current pending mail transaction by sending the mail to the server and makes the session ready for another mail request (equivalent of a protocol <CR><LF>.<CR><LF>).

Refer to section 6.2.1 for more details on the wip_close function.

11. POP3 Client API

The POP3 (Post Office Protocol – Version 3) is a standard protocol for mail retrieving from a mail server by a workstation. It requests using Sierra Wireless TCP/IP implementation (Internet Library). It is based on Internet Library abstract channel interface.

POP3 mail retrieving process is generated in several phases:

- First, the application must create a PÖP3 session/connection channel with the interface wip_POP3ClientCreateOpts() that will store information common to all further POP3 requests: address of the mail server, authentication parameters. This channel will also maintain persistent connections.
- Application should call the wip_listOpts() interface in order to open a list channel. Once the list channel is opened, the wip_read() call will retrieve in a structure the list of all the mail Id and their respective size.
- a DATA channel is then created for each POP3 request using wip_getFile() or wip_getFileOpts()
- wip_read() is then applied to that DATA channel to extract the mail data until WIP_CEV_DONE event indicating that the end of the specified mail is entirely read

11.1. Required Header File

The header file for the POP3 client interface definitions is: wip_pop3.h.

11.2. The Session / Connection Channel

11.2.1. wip_POP3ClientCreateOpts

The wip_POP3ClientCreateOpts allows the application to pass additional configuration options.

Prototype

```
wip_channel_t wip_POP3ClientCreateOpts ( ascii   *server,
                                         wip_eventHandler_f   handler,
                                         void   *ctx,
                                         ... );
```

Parameters

server:

The name of the server: either as a DNS resolved name, or in dotted notation, e.g. “192.168.1.1”.

handler:

The call back handler which receives the network events related to the channel.

The events defined in the table below are supported

Event	Description
WIP_CEV_OPEN	This event is received when the session channel is established
WIP_CEV_DONE	This event is received either when listing mails or deletion of mails is completed.
WIP_CEV_PEER_CLOSE	This event is received when the peer closes down
WIP_CEV_ERROR	This event is received when a socket error or protocol error occurs during the session. An error code can either be positive or a negative value depending on the cause of error. The error code will be negative, if error is due to Internet Library library socket. The error code will be positive, if error is due to POP3 protocol or Internet Library POP3 library. The wip_getOpts function can be used to determine the cause of an error. Refer to the POP3 Error Codes section for error codes.

ctx:

This is the user data to be passed to the event handler every time it is called.

...:

A list of configuration options, the last option must be WIP_COPT_END. Each option can be followed by one value.

Option	Value	Description
WIP_COPT_END	None	This option defines the end of the option list.

Option	Value	Description
WIP_COPT_PEER_PORT	u16	This option sets the port number of the POP3 mail server, the default value is 110.
WIP_COPT_USER	ascii *	This option sets the username, the default value is NULL.
WIP_COPT_PASSWORD	ascii *	This option sets the password, the default value is NULL.
WIP_COPT_FINALIZER	wip_finalizer_f	Function which is called after the channel has been completely closed using wip_close function and all its associated resources have been released.

NOTE: The following options are now available for working with this function as well: WIP_COPT_SND_BUFSIZE, WIP_COPT_RCV_BUFSIZE, WIP_COPT_TOS, WIP_COPT_TTL, WIP_COPT_MAXSEG, WIP_COPT_SND_LOWAT, WIP_COPT_RCV_LOWAT, WIP_COPT_NODELAY and WIP_COPT_KEEPALIVE. Their defaults are the same as the TCP sockets, and these options can be reached through wip_{get,set}Opts.

Returned Value

The function returns

- the created SESSION channel
- else NULL on error

Note: *For the username and password the memory allocations should be managed by the the application.*

11.2.2. The wip_getOpts Function

The wip_getOpts function is used to retrieve options of a SESSION channel. The options supported by the wip_getOpts function, applied to a POP3 Client are:

Option	Value	Comment
WIP_COPT_ERROR	u32*	This event is received when a socket error or protocol error occurs during the session. An error code can either be positive or a negative value depending on the cause of error. The error code will be negative, if error is due to Internet Library socket. The error code will be positive, if error is due to POP3 protocol or Internet Library POP3 library. The wip_getOpts function can be used to determine the cause of an error. default:WM_EOK Refer section 16.5 for error codes.
WIP_COPT_POP3_STATUS_CODE	u32*, ascii **	Returns two arguments: 1) the last protocol error code 2) and the associated response string Refer section 16.5 for error codes.
WIP_COPT_ADDR	ascii **	Get the specified hostname default:0
WIP_COPT_USER	ascii **	Get the specified username
WIP_COPT_PASSWORD	ascii **	Get the specified password
WIP_COPT_PEER_PORT	u32*	Get the specified port default:0
WIP_COPT_POP3_NB_MAILS	u32*	Get the total number of mails (only available after wip_listOpts () is called).
WIP_COPT_POP3_MAILSIZE	u32*	Get the total mail size (only available after wip_listOpts () is called).

Refer section 6.2.6 for more details on the wip_getOpts function.

11.2.3. The wip_deleteFile Function

The wip_deleteFile function is used to mark as deleted the specified mail Id. The mail id to be deleted should be specified in string format. For example, "1","22" etc.

Refer section 7.8 for more details on the wip_deleteFile function.

11.2.4. The wip_close Function

On the SESSION channel the wip_close function aborts any current request and release resources associated with the session channel.

Note: This interface does not close the opened DATA channel. It is the application which is in charge of closing the opened channels.

Refer section 6.2.1 for more details on the wip_close function.

11.3. The List Channel

The LIST channel is used to enumerate and get the listing of all the available mails in the POP3 server.

11.3.1. The wip_listOpts Function

The wip_listOpts function is used open a list channel in order to list all the available mails to be retrieved.

Prototype

```
wip_channel_t wip_listOpts ( wip_channel_t   session,
                             ascii   *name,
                             wip_eventHandler_f    handler,
                             void   *ctx,
                             ... )
```

Parameters

session:

The POP3 SESSION channel

name:

This field is ignored

handler:

The call back handler which receives the events related to the channel.

The events defined in the table below are supported.

Event	Comment
WIP_CEV_OPEN	This event is received when the list channel is established.
WIP_CEV_DONE	This event is received when the mail listing is completed.
WIP_CEV_ERROR	This event is received when a socket error or protocol error occurs during the session. An error code can either be positive or a negative value depending on the cause of error. The error code will be negative, if error is due to Internet Library library socket. The error code will be positive, if error is due to POP3 protocol or Internet Library POP3 library. The wip_getOpts function can be used to determine the cause of an error. Refer section 16.5 for error codes.
WIP_CEV_READ	This event is received when the list channel is ready for read operations to get the mail listing.

ctx:

It is the user data to be passed to the event handler every time it is called.

...:

A list of configuration options, the last option must be WIP_COPT_END. Each option can be followed by one or more values, see wip_POP3ClientCreateOpts () for a description of supported options.

Returned values

The function returns

- OK on success
- else a negative error code

11.3.2. The wip_read Function

The wip_read function is used to read the file structure from the list channel.

Note: The returned file structure is of the type wip_fileInfo_t. Application should manage the memory allocations for the return value of wip_read.

Refer to section 6.2.2 for more details on wip_read function.

11.3.3. The wip_close Function

The wip_close function is used to close the list channel.

Refer to section 6.2.1 for more details on wip_close function.

11.4. The Data Channel

The DATA channel is used in order to retrieve a mail from the POP3 server.

11.4.1. The wip_getFile Function

The wip_getFile function is used to open a DATA channel in order to retrieve a mail.

Following events are supported.

Event	Description
WIP_CEV_OPEN	This event is received when the DATA channel is established and ready for data reading
WIP_CEV_READ	This event is received when mail body data can be read by the application.
WIP_CEV_DONE	This event is received when the entire mail has been read.
WIP_CEV_ERROR	This event is received when a socket error or protocol error occurs during the session. An error code can either be positive or a negative value depending on the cause of error. The error code will be negative, if error is due to Internet Library library socket. The error code will be positive, if error is due to POP3 protocol or Internet Library POP3 library. The wip_getOpts function can be used to determine the cause of an error. Refer section 16.5 for error codes.

Refer section 7.2 for more details on wip_getFile function.

Note: The parameter "filename" is the mail ID in string format. This mail ID has been obtained by listing the mails with the wip_listOpts () interface.

11.4.2. The wip_getFileOpts Function

The wip_getFileOpts allows the application to pass additional configuration options.

The events received are the same as defined for the wip_getFile function in section 11.4.1

The options supported by the wip_getFileOpts function, applied to a POP3Client are:

Option	Value	Description
WIP_COPT_END	none	This option defines the end of the option list.

Refer section 7.3 for more details on wip_getFileOpts function.

11.4.3. The wip_read Function

The wip_read function is used to read the request message body.

Refer section 6.2.2 for more details on wip_read function.

11.4.4. The wip_getOpts Function

The wip_getOpts function is used to retrieves options of a DATA channel.

Data channels support the following options:

Option	Value	Comment
WIP_COPT_ERROR	u32*	This event is received when a socket error or protocol error occurs during the session. An error code can either be positive or a negative value depending on the cause of error. The error code will be negative, if error is due to Internet Library socket. The error code will be positive, if error is due to POP3 protocol or Internet Library POP3 library. The wip_getOpts function can be used to determine the cause of an error. default:WM_EOK Refer section 16.5 for error codes.
WIP_COPT_POP3_MAILSIZE	u32*	Get the mail size of the specified mail ID.

Refer section 6.2.6 for more details on the wip_getOpts function.

11.4.5. The wip_close Function

On a DATA channel the wip_close function closes the channel and makes the session ready for another request.

Refer section 6.2.1 for more details on the wip_close function.

12. MMS Client

MMS client provides an application interface to be able to send MMS using Sierra Wireless TCP/IP implementation (Internet Library). It is based on Internet Library abstract channel interface and HTTP protocol. The following features are provided:

- Send a MMS
- Add different kind of attachments including images or sounds

12.1. Required Header File

The header file for the MMS client interface definition is wip_mms.h.

12.2. The wip_mmsCreate Function

The wip_mmsCreate function creates and initializes a MMS structure.

12.2.1. Prototype

```
wip_mms_t wip_mmsCreate( void );
```

12.2.2. Parameters

None

12.2.3. Returned Values

The function returns the MMS control structure.

12.3. The wip_mmsCreateOpts Function

The function wip_mmsCreateOpts creates and initializes a MMS structure with user defined options.

12.3.1. Prototype

```
wip_mms_t wip_mmsCreateOpts( int   optid1,

                             ... );
```

12.3.2. Parameters

`optid1:`

In: One of the options from the list below. If no option is provided then at least WIP_COPT_END must be specified.

`...:`

In: These parameters are a list of configuration options followed by one or more option values. The last option must be WIP_COPT_END. The supported options and their associated values are defined in the table below.

Option	Description	Set Type	Get Type
WIP_COPT_END	Indicates the end of the option list.	<none>	<none>
WIP_COPT_MMS_DATE	This option gives the date of the MMS (in seconds). See the example for MMS i.e. section 15.10 for more details.	<u32>	<u32*>
WIP_COPT_MMS_TO_PHONE	Recipient of the MMS (recipient is a phone number. Length of the field must not exceed 50 characters). This recipient will be added at the “To” list.*	<ascii *>	not applicable
WIP_COPT_MMS_CC_PHONE	Recipient of the MMS (recipient is a phone number. Length of the field must not exceed 50 characters). This recipient will be added at the “Cc” list. *	<ascii *>	not applicable
WIP_COPT_MMS_BCC_PHONE	Recipient of the MMS (recipient is a phone number. Length of the field must not exceed 50 characters). This recipient will be added at the “Bcc” list. *	<ascii *>	not applicable
WIP_COPT_MMS_TO_MAIL	Recipient of the MMS (recipient is a mail address. Length of the field must not exceed 50 characters). This recipient will be added at the “To” list.*	<ascii *>	not applicable
WIP_COPT_MMS_CC_MAIL	Recipient of the MMS (recipient is a mail address. Length of the field must not exceed 50 characters). This recipient will be added at the “Cc” list. *	<ascii *>	not applicable

Option	Description	Set Type	Get Type
WIP_COPT_MMS_BCC_MAIL	Recipient of the MMS (recipient is a mail address. Length of the field must not exceed 50 characters). This recipient will be added at the “Bcc” list. *	<ascii *>	not applicable
WIP_COPT_MMS_SUBJECT	Subject of the MMS (length of the field must not exceed 100 characters).	<ascii *>	<ascii *>
WIP_COPT_MMS_MESSAGE_CLASS	Class of the message.	WIP_MMS_MESSAGE_PERSONAL WIP_MMS_MESSAGE_INFORMATIONAL WIP_MMS_MESSAGE_ADVERTISEMENT WIP_MMS_MESSAGE_AUTO	<u32*>
WIP_COPT_MMS_PRIORITY	Priority of the message.	WIP_MMS_PRIORITY_LOW, WIP_MMS_PRIORITY_NORMAL, WIP_MMS_PRIORITY_HIGH	<u32*>
WIP_COPT_MMS_SENDER_VISIBILITY	Visibility of the sender.	WIP_MMS_SENDER_SHOW WIP_MMS_SENDER_HIDE	<u32*>
WIP_COPT_MMS_FROM	Sender of the MMS (length of the field must not exceed 50 characters).	<ascii *>	<ascii *>
WIP_COPT_MMS_MULTIPART_TYPE	First parameter is the multipart type of the MMS In case of WIP_MMS_MULTIPART_RELATED: Second parameter is the presentation file type (Type field),(less than 50 characters) Third parameter is start file identification (Start field)(less than 50 characters)	WIP_MMS_MULTIPART_MIXED (default value) or WIP_MMS_MULTIPART_RELATED (in case of presentation file included), <string>,< string>	<u32*>, <string>, <string>
WIP_COPT_MMS_STATUS	Callback called when the MMS was sent (gives the status of the sent item). First parameter is the pointer on the callback function. Second parameter is a pointer to the context passed to the callback.	<statusCb_pf>,<void * ctx> [typedef void (*statusCb_pf) (wip_mms_t mmsCtrl, u32 status, void *ctx);]	<statusCb_pf>

* The total number of recipients (To+Cc+Bcc) must be 12 or less, and for each recipient list (To or Cc or Bcc), the string length must be less than 250 characters including "/TYPE=PLMN" in case of phone recipient type.Returned Value

The function returns

- MMS control structure on success
- NULL if a problem occurred

Note: *At least one of the following address fields must be present:*
WIP_COPT_MMS_TO_PHONE
WIP_COPT_MMS_TO_MAIL
WIP_COPT_MMS_CC_PHONE
WIP_COPT_MMS_CC_MAI
WIP_COPT_MMS_BCC_PHONE
WIP_COPT_MMS_BCC_MAIL

Note: *Options WIP_COPT_MMS_FROM and WIP_COPT_MMS_SUBJECT are mandatory.*

The option WIP_COPT_MMS_MESSAGE_CLASS only supports WIP_MMS_MESSAGE_PERSONAL for many operators.

The option WIP_COPT_MMS_PRIORITY doesn’t support WIP_MMS_PRIORITY_HIGH.

12.4. The status callback function

This is the callback handler which is called when the MMS is sent.

12.4.1. Prototype

```
typedef void( *statusCb_pf ) ( wip_mms_t mmsCtrl, u32 status, void *ctx );
```

12.4.2. Parameters

`mmsCtrl:`

In: MMS control structure.

`status:`

In: The possible values are:

Status values	Description
WIP_MMS_STATUS_OK	Status is OK
WIP_MMS_STATUS_SERVICE_DENIED	Service denied
WIP_MMS_STATUS_MESSAGE_FORMAT_CORRUPT	Message format corrupted
WIP_MMS_STATUS_SENDING_ADDRESS_UNRESOLVED	Sending address is unresolved
WIP_MMS_STATUS_MESSAGE_NOT_FOUND	Message is not found
WIP_MMS_STATUS_NETWORK_PROBLEM	Some network problem
WIP_MMS_STATUS_CONTENT_NOT_ACCEPTED	Content is not accepted
WIP_MMS_STATUS_UNSUPPORTED_MESSAGE	Message is not supported
WIP_MMS_STATUS_UNSPECIFIED_ERROR	Unspecified error

`ctx:`

In/out: Pointer on the context previously passed with the pointer on the callback.

12.5. The wip_mmsSetOpts Function

The wip_mmsSetOpts function is used to change options values in the MMS Control structure.

12.5.1. Prototype

```
int wip_mmsSetOpts( wip_mms_t   mmsCtrl,

                    int   optid1,

                    ... );
```

12.5.2. Parameters

`mmsCtrl:`

In/out: MMS control structure.

`optid1:`

In: One of the options listed in the section 13.3.2. If no option is provided then at least WIP_COPT_END must be specified.

`...:`

In: These parameters are a list of configuration options followed by one or more option values. The last option must be WIP_COPT_END. For the supported options and their associated values refer to the section 12.3.2.

12.5.3. Returned Values

The function returns

- WIP_CERR_OK on success
- WIP_CERR_INTERNAL if a problem occurred

12.6. The wip_mmsGetOpts Function

The wip_mmsGetOpts function is used to get options values in the MMS Control structure.

12.6.1. Prototype

```
int wip_mmsGetOpts( wip_mms_t   mmsCtrl,

                    int   optidl,

                    ... );
```

12.6.2. Parameters

mmsCtrl:

In/out: MMS control structure.

optidl:

In: One of the options listed in the section 13.3.2. If no option is provided then at least WIP_COPT_END must be specified.

...:

Out: These parameters are a list of configuration options followed by one or more option values. The last option must be WIP_COPT_END. For the supported options and their associated values refer to the section 12.3.2 except for the two options WIP_COPT_MMS_TO_PHONE and WIP_COPT_MMS_TO_MAIL which are replaced by the unique option WIP_COPT_MMS_TO and WIP_COPT_MMS_CC_PHONE and WIP_COPT_MMS_CC_MAIL replaced by the unique option WIP_COPT_MMS_CC and WIP_COPT_MMS_BCC_PHONE and WIP_COPT_MMS_BCC_MAIL replaced by the unique option WIP_COPT_MMS_BCC.

12.6.3. Returned Values

The function returns

- WIP_CERR_OK on success
- WIP_CERR_INTERNAL if a problem occurred

12.7. The wip_mmsAddPart Function

The wip_mmsAddPart function adds an attachment on the MMS.

12.7.1. Prototype

```
int wip_mmsAddPart( wip_mms_t   mmsControl,

                    u8   *ptrToAttachement,

                    u32   sizeOfAttachment,

                    int   optid1,

                    ... );
```

12.7.2. Parameters

mmsCtrl:

In/out: MMS control structure.

ptrToAttachment:

In: pointer on the attachement.

sizeOfAttachment:

In: size of the attachement.

optid1:

In: One of the options from the list below. If no option is provided then at least WIP_COPT_END must be specified.

...:

In: These parameters are a list of configuration options followed by one or more option values. The last option must be WIP_COPT_END.The supported options and their associated values are defined in the table below.

Option	Value	Description
WIP_COPT_END	none	Indicates the end of the option list.
WIP_COPT_MMS_PART_TEXT	none	The part that we add is a text in US-ASCII format.
WIP_COPT_MMS_PART_JPG	<u8 *>, <u32>	The part that we add is a jpg image. First parameter is the name of the picture. The second parameter is the length of the name of the picture.

Option	Value	Description
WIP_COPT_MMS_PART_AMR	<u8 *>, <u32>	The part that we add is an AMR sound. First parameter is the name of the sound. The second parameter is the length of the name of the sound.
WIP_COPT_MMS_PART_TEXT_UTF8	none	The part that we add is a text in UTF-8 format.
WIP_COPT_MMS_PART_TEXT_UTF16	none	The part that we add is a text in UTF-16 format.
WIP_COPT_MMS_PART_TEXT_UCS2	none	The part that we add is a text in UCS-2 format.
WIP_COPT_MMS_PART_TEXT_US_ASCII	none	The part that we add is a text in US_ASCII format.
WIP_COPT_MMS_PART_JPEG	none	The part that we add is a jpeg or jpg image.
WIP_COPT_MMS_PART_GIF	none	The part that we add is a gif image.
WIP_COPT_MMS_PART_TIFF	none	The part that we add is a png image.
WIP_COPT_MMS_PART_PNG	none	The part that we add is a png image.
WIP_COPT_MMS_PART_WBMP	none	The part that we add is a vnd.wap.wbmp image.
WIP_COPT_MMS_PART_SMIL	none	The part that we add is a smil presentation file.
WIP_COPT_MMS_PART_ANY	string	The part that we add is a file of any type. The parameter is the Content-Type name (must not exceed 50 characters)
WIP_COPT_MMS_PART_ID	string	Give the part identification The parameter is the Content-ID of the part (must not exceed 50 characters)) format: “<XXX>” useful for multipart/related MMS
WIP_COPT_MMS_PART_NAME	string	Give the part file name The parameter is the Content-Location of the part (must not exceed 50 characters)

Note: *In case of option WIP_COPT_MMS_PART_TEXT, neither file name nor ID can be given.*

Note: *To have the following header:*

Content-Type: image/jpeg
Content-Transfer-Encoding: base64
Content-ID: <001>
Content-Location: picture.jpg

the call is as follows:

```
wip_mmsAddPart(mmsControl, ptrToAttachement, SizeOf Attachement,
WIP_COPT_MMS_PART_JPEG, WIP_COPT_MMS_PART_NAME", "picture.jpg,
WIP_COPT_MMS_PART_ID, "<001>")
```

Note: *To have the following header:*

Content-Type: image/jpeg; name=toto.jpg
Content-Transfer-Encoding: base64
Content-ID: <001>
Content-Location: picture.jpg

the call is as follows:

```
wip_mmsAddPart(mmsControl, ptrToAttachement, SizeOf Attachement,
WIP_COPT_MMS_PART_ANY,  "image/jpeg;
name=toto.jpg",WIP_COPT_MMS_PART_NAME", "picture.jpg, WIP_COPT_MMS_PART_ID,
"<001>")
```

12.7.3. Returned Value

The function returns

- WIP_CERR_OK on success
- WIP_CERR_INVALID if a problem occurred

Note: *Option(WIP_COPT_MMS_PART_JPG should not be used anymore*

In case of an AMR part, whatever the name specified, it will be replaced by “sound.amr”.

Attachments must be added one by one (if you want to add 3 parts, you need to call the function three times).

Each attachment size should not exceed 2MB.

12.8. The wip_mmsRemovePart Function

The wip_mmsRemovePart function removes an attachment previously attached on the MMS.

12.8.1. Prototype

```
int wip_mmsRemovePart( wip_mms_t   mmsControl,

                       u8   *ptrToAttachement )
```

12.8.2. Parameters

`mmsCtrl:`

In/out: MMS control structure.

`ptrToAttachement:`

In: pointer to the attachment that needs to be removed.

Note: *Only one attachment can be removed at a time For eg. If three parts attached on the MMS need to be removed then this function should be called three times.*

12.8.3. Returned Values

The function returns

- WIP_CERR_OK on success
- WIP_CERR_INVALID if the attachment is not found.

12.9. The wip_mmsSend Function

The wip_mmsSend function is used to send the MMS once all the attachments have been added.

12.9.1. Prototype

```
int wip_mmsSend( wip_mms_t   mmsControl,

                 wip_channel_t   HTTPCnxChannel,

                 u8   *HttpUrl,

                 int   optid,

                 ... );
```

12.9.2. Parameters

mmsControl:

In/out: MMS control structure.

HTTPCnxChannel:

In: HTTP connection channel.

HttpUrl:

In: URL of the MMS server.

optid:

In: No option exists.

...:

In/out: No option exists.

Note: If the mandatory parameters are not mentioned, the sending of the MMS aborts and returns an error.

12.9.3. Returned Values

The function returns

- WIP_CERR_OK on success
- WIP_CERR_INTERNAL if the HTTP data channel cannot be created or MMS total size exceeds 300Kb

12.10. The wip_mmsClose function

The wip_mmsClose function is used to destroy MMS structure and release memory.

12.10.1. Prototype

```
int wip_mmsClose( wip_mms_t   mmsControl )
```

12.10.2. Parameters

`mmsControl:`

In/out: MMS control structure.

12.10.3. Returned Values

The function returns

- WIP_CERR_OK on success

WIP_CERR_INTERNAL if the mmsControl structure is NULL.

13. SNMP client API

SNMP client provides an application interface to add SNMP agent using Sierra Wireless TCP/IP implementation (Internet Library). It is based on Internet Library abstract channel interface. The following features are provided:

- provide standard SNMP client (standard MIB)
- create/remove MIB modules
- send trap messages

13.1. Required Header File

The header file for the SNMP client interface definition is wip_snmp.h.

13.2. The WIPmibcol_S Structure

The WIPmibcol_S structure defines the structure for columns in MIB table (sequence MIB entries).

```
typedef struct WIPmibcol_S {
  u8 mc_oid;        /* oid of the column (column identifier) */
  u8 mc_type[2];    /* type of value*/
} WIPmibcol;
```

13.3. The WIPmibent_S Structure

The WIPmibent_S structure defines the structure for MIB entries in MIB modules.

```
typedef struct WIPmibent_S {
  u8 me_oid;                      /* base MIB module object identifier */
  u8 me_type[2];                  /* type of value
                                  [0]:asn or snmp type
                                  [1]:opaque type */
  WIPmibcol const *me_cols;       /* list of columns (index one first)*/
  u32 me_ncols;                   /* number of columns in previous list*/
  u32 me_nidx;                    /* number of index columns in previous
                                     list */
} WIPmibent;
```

Note: Elements me_cols, me_ncols and me_nidx of the structure are valid only if me_type[0] = WIP_ASN_SEQUENCE.

13.4. The WIPmibmod_S Structure

The WIPmibmod_S structure defines the structure for MIB modules.

```
typedef struct WIPmibmod_S {
  u8 mm_oid[WIP_MIBID_MAX_LEN];     /* base MIB module object identifier */
  u32 mm_oidlen;                    /* OID length of the MIB module */
  const WIPmibent *mm_ent;          /* points to first entity */
  u32 mm_nent;                      /* number of entries in MIB */
  int(* mm_cb_f) (                  /* MIB access function (callback) */
  struct WIPmibparamcb_S *arg,
  u32 *val,
  u32 *vallen,
  u32 cbdata);
  u32 mm_cbdata;                    /* callback data for arg cbdata of
                                       mm_cb_f */
} WIPmibmod;
```

13.5. The WIPmibparamcb_S Structure

The WIPmibparamcb_S structure defines the parameters of MIB module callbacks.

```
typedef struct WIPmibparamcb_S {
  struct WIPsnmpsession_S *session;  /* pointer to SNMP session */
  u8 cmd;                            /* SNMP command
                                        For eg: get, getnext, check/set */
  struct WIPmibmod_S const *mib;     /* current MIB module */
  u32 mibdata;                       /* MIB callback data */
  struct WIPmibent_S const *mibent;  /* current MIB entry */
  struct WIPmibcol_S const *mibcol;  /* current column of mibent (for
                                        sequence) */
  u8 const *oid;                     /* pointer to read OID */
  u32 oidlen;                        /* size of OID */
  u32 nidx;                          /* number of index */
  u32 idx[WIPSNMP_MAX_IDX];          /* index values */
  u32 idxlen[WIPSNMP_MAX_IDX];       /* length of the values */
} WIPmibparamcb;
```

Note: WIPSNMP_MAX_IDX is equal to 8.

13.6. The wip_snmpInitOpts Function

The function wip_snmpInitOpts initializes all SNMPv{1,2,3} standard MIB given below:

- MibSystem
- MibSnmp
- MibSnmpSet
- MibSnmpEngine
- MibSnmpTarget
- MibUsmStats
- MibUsmUser
- MibVacm
- MibInterfaces
- MibAt
- MibIp
- MibIcmp
- MibUdp
- MibTcp

13.6.1. Prototype

```
int wip_snmpInitOpts( u32   optid1,

                               ... );
```

13.6.2. Parameters

optid1:

In: One of the options from the list below.If no option is provided then at least WIP_COPT_END must be specified.

...:

In: These parameters are a list of configuration options followed by one or more option values. The last option must be WIP_COPT_END. The supported options and their associated values are defined in the table below.

Option	Value Type	Description
WIP_COPT_END	none	Indicates the end of the option list.
WIP_COPT_ACCESS	defined below	Defined below.
WIP_COPT_GROUP	defined below	Defined below.
WIP_COPT_COMMUNITY	defined below	Defined below.
WIP_COPT_VIEW	defined below	Defined below.
WIP_COPT_CONTEXT	defined below	Defined below.
WIP_COPT_TRAP	defined below	Defined below.
WIP_COPT_USER	defined below	Defined below.

Option	Value Type	Description
WIP_COPT_ENGINE_ID	<u8 *engineid>, <u32 engineid_length>	Unique identifier of the SNMP agent. This entry is mandatory
WIP_COPT_TRAP_COM	<ascii *>	Name of the default community for traps.
WIP_COPT_SYS_CONTACT	<ascii *>	Content of the SNMP data sysContact. sysContact is an information which can be retrieved through SNMP walker. See a SNMP protocol doc for more information.
WIP_COPT_SYS_LOCATION	<ascii *>	Content of the SNMP data sysLocation. sysLocation is an information which can be retrieved through SNMP walker. See a SNMP protocol doc for more information.
WIP_COPT_SND_BUFSIZE	<u32>	Size of the emission buffer.
WIP_COPT_SYS_SERVICES	<u32>	Content of the SNMP data sysServices. sysServices is an information which can be retrieved through SNMP walker. See a SNMP protocol doc for more information.
WIP_COPT_GROUPS_BUFSIZE	<u32>	Number of groups that can be created dynamically through SNMP queries.
WIP_COPT_SYS_OID	<u8 *iod>, <u32 oid_length>	Content of the SNMP data sysObjectID. sysObjectID is an information which can be retrieved through SNMP walker. See a SNMP protocol doc for more information.
WIP_COPT_SYS_DESCR	<ascii *>	Content of the SNMP data sysDescr. sysDescr is an information which can be retrieved through SNMP walker. See a SNMP protocol doc for more information.
WIP_COPT_SYS_NAME	<ascii *>	Content of the SNMP data sysName. sysName is an information which can be retrieved through SNMP walker. See a SNMP protocol doc for more information.
WIP_COPT_RCV_BUFSIZE	<u32>	Size of the reception buffer.
WIP_COPT_BOOTS	<u32>	Number of reboots experienced by the SNMP engine. This is mainly used for protection against replay attacks.
WIP_COPT_AUTH_TRAPS	<u32>	Content of the SNMP data snmpEnableAuthenTraps. snmpEnableAuthenTraps is an information which can be retrieved through SNMP walker. See a SNMP protocol doc for more information.
WIP_COPT_USERS_BUFSIZE	<u32>	Number of users that can be created dynamically through SNMP queries.

There are several complex options which are explained along with the option value and description in detail below:

WIP_COPT_ACCESS:

- <ascii *group_name>: Name of the group whose access rights are being modified.
- <ascii *context>: Context in which this modification is relevant.
- <u32 is_context_prefix>: When set to true, all contexts whose name start with the previous parameter are modified, rather than on the context whose name exactly matches it.
- <u32 sec_model>: Security model associated with these access rights. It can be one of the following:

  - 0 (any model accepted),
  - 1 (SNMPv1),
  - 2 (SNMPv2), or
  - 3 (SNMPv3)
- <u32 sec_level>: Security level associated with these access rights. It can be one of the following:
  - WIP_SNMP_SECLVL_NOAUTH: No authentication
  - WIP_SNMP_SECLVL_AUTHNOPRIV: Authentication, no privacy
  - WIP_SNMP_SECLVL_AUTHPRIV: Authentication and privacy
- <ascii *read>: Name of a view: all nodes of this view are accessible for reading with these access rights. It can be set to NULL.
- <ascii *write>>: Name of a view: all nodes of this view are accessible for writing with these access rights. It can be set to NULL.
- <ascii *notify>>: Name of a view: all nodes of this view are accessible for notification with these access rights. It can be set to NULL.

WIP_COPT_GROUP:

- <ascii *group_name>: Name of a group.
- <u32 snmp_version>: SNMP version, either 1, 2 or 3.
- <ascii *user_name>: Name of a user which is included in the group group_name.

This option permits to add a user to a group. There can be several WIP_COPT_GROUP declarations per group, and several groups defined.

WIP_COPT_COMMUNITY:

- <ascii *user_name>: Name of the user to which the community will be mapped.
- <ascii *straddr>: Address of the community's network. No DNS request allowed, i.e. the address must have the numeric form "nnn.nnn.nnn.nnn".
- <ascii *netmask>: Netmask of the network, e.g. "255.255.255.0".
- <ascii *community_name>: Name of the community. Most often same as the user_name.

SNMPv{1,2} work with communities, rather than users, groups and views as does SNMPv3. In order to allow SNMPv3 to handle SNMPv{1,2} requests, this WIP_COPT_COMMUNITY entry allows to map v{1,2} communities to v3 users. It is not necessary to declare these community users in a WIP_COPT_USER entry.

WIP_COPT_VIEW:

- <ascii *name>: The view's name.
- <u32 exclude>: A boolean: When set to TRUE the subtree is removed from the view.When set to FALSE subtree is added to the view.
- <u8 *oid>: The object id giving the root of the subtree which should be added/removed from the view.
- <u8 *oidmask>: A bitfield indicating which part of the oids is relevant.
- <u32 oid_length>: Length of the oid. The mask's length is deduced from this, it is oid_length/8.

In SNMPv3, access rights are attached to subsets of the base and these subsets are described by views. Each WIP_COPT_VIEW entry alters a subset by adding or removing subtrees from it.

WIP_COPT_CONTEXT:

- <ascii *>: Context supported by the SNMP agent. There can be several of them in the configuration, each one introduced by its own WIP_COPT_CONTEXT option id. If no context is provided, the default empty context "" is assumed.

WIP_COPT_TRAP:

- <u32 version>: SNMP protocol version in which the trap will be served. It can be 1, 2 or 3.
- <ascii *addr>: Destination address of the TRAP. No DNS request allowed, i.e. the address must have the numeric form “nnn.nnn.nnn.nnn”.
- <u32 port>: Destination TCP port.
- <ascii *security>: Security name.
- <u32 security_level>: Security level. It can be one of the following:
  - WIP_SNMP_SECLVL_NOAUTH: No authentication and no privacy
  - WIP_SNMP_SECLVL_AUTHNOPRIV: Authentication, but no privacy
  - WIP_SNMP_SECLVL_AUTHPRIV: Both authentication and privacy

WIP_COPT_USER:

- <ascii *name>: The user’s security name.
- <u32 auth_scheme>: Authentication scheme, one of the following:
  - WIP_SNMP_AUTH_NONE: No authentication
  - WIP_SNMP_AUTH_MD5: Authentication through MD5 HMAC
  - WIP_SNMP_AUTH_SHA: Authentication through SHA HMAC
- <u32 priv_scheme>: Privacy scheme. It can be one of the following:
  - WIP_SNMP_PRIV_NONE: No privacy
  - WIP_SNMP_PRIV_DES: Privacy through DES encryption. It is illegal to provide encryption without authentication, so the combination WIP_SNMP_AUTH_NONE, WIP_SNMP_PRIV_DES will cause an error
- <ascii *password>: Password used, together with the engine ID, to create the localized MD5 or SHA key, used for authentication and encryption. This field can be left NULL if authentication is disabled with WIP_SNMP_AUTH_NONE.

This option permits to declare a new user, with the security level settings. All users handled by SNMP must be declared, either here, or with WIP_COPT_COMMUNITY for SNMPv{1,2}.

There can be several user declarations in the configuration, each one introduced by its own WIP_COPT_USER option identifier.

13.6.3. Returned Values

The function returns

- OK on success
- WM_EACCES if the TCP/IP feature is not active
- WM_ALREADY if the SNMP agent is already running

13.7. The wip_snmpClose Function

The wip_snmpClose function closes the SNMP agent.

13.7.1. Prototype

```
int wip_snmpClose( void );
```

13.7.2. Parameters

None

13.7.3. Returned Values

The function returns WM_EOK on success.

13.8. The wip_snmpv3Trap Function

The wip_snmpv3Trap function is used when the agent wants to inform all trap receivers about the value of some variables in a MIB module. The function allows the application to give more details by sending some {OID, value} pairs encoded in ASN.1 using the arguments vars and varslen. If the receiver is a SNMPv1 manager, the arguments gentrap and spectrap values are used. If the receivers are SNMPv2c or SNMPv3 then trapoid and trapoidlen values are used.

13.8.1. Prototype

```
int wip_snmpv3Trap( u32   gentrap,

                    u32   spectrap,

                    const u8   *trapoid,

                    u32   trapoidlen,

                    const void   *vars,

                    u32   varslen );
```

13.8.2. Parameters

`gentrap`:

Generic trap type. (SNMPv1 Trap)

`spectrap`:

Specific trap code (SNMPv1 Trap).

`trapoid`:

Trap OID (SNMPv2c and SNMPv3 Trap).

`trapoidlen`:

Length of Trap OID (SNMPv2c and SNMPv3 Trap).

`vars`:

Pointer to additional variables bindings.

`varslen`:

Length of additional variables.

13.8.3. Returned Values

The function returns

- WM_EOK on success
- WM_EINVAL if an argument is invalid
- WM_ENOSPC if there is no enough space to create the PDU

13.9. The wip_snmpv3TrapTo Function

The wip_snmpv3TrapTo function is used when the agent wants to inform a SNMP manager about the value of certain variables in a MIB module. The function allows the application to give more details by sending some {OID, value} pairs encoded in ASN.1 using the arguments vars and varslen. The destination address may contain a port number value of 0, in this case the default SNMP trap port number is used. When sending a SNMPv1 trap the gentrap and spectrap argument values must be provided, the trapoid and trapoidlen arguments are ignored. When sending a SNMPv2c or SNMPv3 trap the trapoid and trapoidlen values are used, the gentrap and spectrap arguments are ignored.

13.9.1. Prototype

```
int wip_snmpv3TrapTo( const wm_sockAddr_in_t *addr,

                      u32   version,

                      const char   *security,

                      u32   seclevel,

                      u32   gentrap,

                      u32   spectrap,

                      const u8   *trapoid,

                      u32   trapoidlen,

                      const void   *vars,
                      u32   varslen );
```

13.9.2. Parameters

addr:

Destination address buffer.

version:

SNMP version: It can be one of the following:

- WM_SNMP_VERSION_V1,
- WM_SNMP_VERSION_V2C or
- WM_SNMP_VERSION_V3

security:

Community (SNMPv1/v2c Trap) or user name (SNMPv3).

level:

Security level (SNMPv3).

`gentrap:`

Generic trap type (SNMPv1 Trap).

`spectrap:`

Specific trap code (SNMPv1 Trap).

`trapoid:`

Trap OID (SNMPv2c and SNMPv3 Trap).

`trapoidlen:`

Length of Trap OID (SNMPv2c and SNMPv3 Trap).

`vars:`

Pointer to additional variables bindings.

`varslen:`

Length of additional variables.

13.9.3. Returned Values

The function returns

- WM_EOK on success
- WM_EINVAL if an argument is invalid
- WM_ENOSPC if there is no enough space to create the PDU

13.10. The wip_snmpMibAdd Function

The wip_snmpMibAdd function is used to add a new MIB module in the SNMP agent. The function checks if the module OID does not already exist.

13.10.1. Prototype

```
int wip_snmpMibAdd( WIPmibmod const   *mib);
```

13.10.2. Parameters

mib:

MIB module which needs to be added.

13.10.3. Returned Values

The function returns

- WM_EOK on success
- WM_EINVAL if the MIB module is not properly initialised

13.11. The wip_snmpMibAddEx Function

The wip_snmpMibAddEx function is used to add a new MIB in the SNMP agent. The function checks if the module OID does not already exist.

13.11.1. Prototype

```
int wip_snmpMibAddEx( WIPmibmod const   *mib,

                      u32   mibdata )
```

13.11.2. Parameters

`mib:`

MIB module which needs to be added

`mibdata:`

data passed to MIB callback function

13.11.3. Returned Values

The function returns

- WM_EOK on success
- WM_EINVAL if the MIB module is not properly initialised

13.12. The wip_snmpMibRemove Function

The wip_snmpMibRemove function is used to remove a MIB module of the SNMP agent.

13.12.1. Prototype

```
int wip_snmpMibRemove( WIPmibmod const   *mib );
```

13.12.2. Parameters

mib:

MIB module which needs to be removed.

13.12.3. Returned Values

The function returns

- WM_EOK on success
- WM_EINVAL if the MIB module does not exist

13.13. The wip_snmpTrap Function

The wip_snmpTrap function is used when the agent wants to inform all trap receivers about the value of some variables in a MIB module. The function allows the application to give more details by sending some {OID, value} pairs encoded in ASN.1 using the arguments vars and varslen. If the receiver is a SNMPv1 manager, the gentrap and spectrap argument values are used. If the receiver is SNMPv2c, trapoid and trapoidlen arguments values are used.

13.13.1. Prototype

```
int wip_snmpTrap( u32   gentrap,

                  u32   spectrap,

                  const u8   *trapoid,

                  u32   trapoidlen,

                  const void   *vars,

                  u32   varslen );
```

13.13.2. Parameters

gentrap:

Generic trap type (SNMPv1 Trap).

spectrap:

Specific trap code (SNMPv1 Trap).

trapoid:

Trap OID (SNMPv2c Trap).

trapoidlen:

Length of Trap OID (SNMPv2c Trap).

vars:

Pointer to additional variables bindings.

varslen:

Length of additional variables.

13.13.3. Returned Values

The function returns

- WM_EOK on success
- WM_EINVAL if an argument is invalid
- WM_ENOSPC if there is no enough space to create the PDU

13.14. The wip_snmpTrapTo Function

The wip_snmpTrapTo function is used when the agent wants to inform a SNMP manager about the value of certain variables in a MIB module. The function allows the application to give more details by sending some {OID, value} pairs encoded in ASN.1 using the arguments vars and varslen. The destination address may contain a port number value of 0, in this case the default SNMP trap port number is used. When sending a SNMPv1 trap the gentrap and spectrap argument values must be provided, the trapoid and trapoidlen arguments are ignored. When sending a SNMPv2c Trap, the trapoid and trapoidlen values are used, the gentrap and spectrap arguments are ignored.

13.14.1. Prototype

```
int wip_snmpTrapTo( const wm_sockAddr_in_t *addr,

                    u32   version,

                    const char   *com,

                    u32   gentrap,

                    u32   spectrap,

                    const u8   *trapoid,

                    u32   trapoidlen,

                    const void   *vars,

                    u32   varslen );
```

13.14.2. Parameters

addr:

Destination address buffer.

version:

SNMP version: It can be one of the following:

- WM_SNMP_VERSION_V1 or
- WM_SNMP_VERSION_V2C

com:

Community string.

gentrap:

Generic trap type (SNMPv1 Trap).

spectrap:

Specific trap code (SNMPv1 Trap).

trapoid:

Trap OID (SNMPv2 Trap).

trapoidlen:

Length of Trap OID (SNMPv2 Trap).

vars:

Pointer to additional variables bindings.

varslen:

Length of additional variables.

13.14.3. Returned Values

The function returns

- WM_EOK on success
- WM_EINVAL if an argument is invalid
- WM_ENOSPC if there is no enough space to create the PDU

14. Multitasking Feature

From OS version 6.00 onwards, an application can define several tasks and build an application using multitasking properties. For more details about the multitasking feature of ADL Library, please refer to the ADL user guide.

The Internet Library also can be used in multitasking manner where different Internet Library can be done from different task context. Note that the bearer management must be done in the main application task context as it is not possible to manage bearers outside the main application task. But socket/session related operations can be done from other tasks contexts too. Also the “wip_netinit” API has to be called from each task that would need any IP communication library service to reserve the associated execution context for each Internet Library operation.

15. Practical Examples

15.1. Initializing a GPRS Bearer

```
#include <wip_bearer.h>
/* bearer events handler */
void myHandler( wip_bearer_t br, s8 event, void *context)
{
  switch( event) {
  case WIP_BEV_IP_CONNECTED:
    /*IP connectivity we can start IP application from here*/
    break;
  case WIP_BEV_IP_DISCONNECTED:
    /*stop IP application*/
    break;
  /* other events: */
  default:
    /*cannot start bearer: report error to higher levels*/
    break;
  }
}
/* bearer handle */
wip_bearer_t myBearer;

/* initialize and start GPRS bearer */
bool myConnectToGPRS( void)
{
  /* open bearer and install our event handler */
  if( wip_bearerOpen( &myBearer, "GPRS", myHandler, NULL) != 0) {
    /* cannot open bearer */
    return FALSE;
  }

  /* configure GPRS interface */
  if( wip_bearerSetOpts ( myBearer,
                          WIP_BOPT_GPRS_APN,      "my_apn",
                          WIP_BOPT_ LOGIN,    "my_login",
                          WIP_BOPT_ PASSWORD, "my_password",
                          WIP_BOPT_END) != 0) {
    /* cannot configure bearer */
```

```
    wip_bearerClose( myBearer);
    return FALSE;
  }

  /* start connection */
  if( wip_bearerStart( myBearer) != 0) && (wip_bearerStart( myBearer) !=
WIP_BERR_OK_INPROGRESS)) {
    /* cannot start bearer */
    wip_bearerClose( myBearer);
    return FALSE;
  }

  /* connection status will be reported to the event handler */
  return TRUE;
}
```

15.2. Simple TCP Client/Server

In this example, the server can receive requests “name”, “forename”, or “phone”, and will answer with the appropriate identification string. It can also receive “quit”, in which case it sends a farewell message and closes the connection.

15.2.1. Server

```
#define SERVER_PORT 1234

#define MSG_WELCOME      "Hello"
#define MSG_SYNTAX_ERROR "Unrecognized request.”\
"Use one of NAME, FORENAME, PHONE, QUIT.\n"

#define MY_NAME     "Adam"
#define MY_FORENAME "User"
#define MY_NAME     "+33 46 29 40 39"

void commHandler( wip_event_t *ev, void *ctx) {
  u8 *buffer[16];
  s32 nread;
  wip_channel_t c = ev->channel;

  switch( ev->kind) {

  case WIP_CEV_OPEN:
    wip_write( c, MSG_WELCOME, strlen( MSG_WELCOME);
    break;

  case WIP_CEV_READ:
    nread = wip_read( c, buffer, sizeof( buffer));
    if( !strncasecmp( buffer, "name", nread))
      wip_write( c, MY_NAME, strlen( MY_NAME));
    else if( !strncasecmp( buffer, "forename", nread))
      wip_write( c, MY_FORENAME, strlen( MY_FORENAME));
    else if( !strncasecmp( buffer, "phone", nread))
      wip_write( c, MY_PHONE, strlen( MY_PHONE));
    else if( !strncasecmp( buffer, "quit", nread))
      wip_close( c);
    else
      wip_write( c, MSG_SYNTAX_ERROR, strlen( MSG_SYNTAX_ERROR));
    return;
```

```
  case WIP_CEV_WRITE:
  case WIP_CEV_ERROR:
  case WIP_CEV_PEER_CLOSE:
    return;
  }
}

void initServer() {
  wip_channel_t server = wip_TCPServerCreate( SERVE_PORT_NUMBER,
                                              &commHandler, NULL);
}
```

15.2.2. Client

The client will request, receive and display the forename, name and phone from the server, then quit by sending the “quit” request to the server. The state of the client is maintained by an enum state as the event handler’s context.

Maintaining the state through a state machine is quite typical of callback-based applications. In a multi-threaded application, the thread is maintained by putting the threads in idle mode and reviving them when an event occurs to them. Here, the event handler is called, from its first line, each time an event happens. The state can be used to remember what has already been done, and what the next thing to do is.

```
#define SERVER_PORT 1234
#define SERVER_ADDRESS "192.168.1.4"

enum state {
  JUST_OPEN,
  FORENAME_REQUEST_SENT,
  NAME_REQUEST_SENT,
  PHONE_REQUEST_SENT,
  QUIT_REQUEST_SENT };

void commHandler( wip_event_t *ev, enum state *ctx) {
  u8 *buffer[256];
  s32 nread;
  wip_channel_t c = ev->channel;
  switch( ev->kind) {
  case WIP_CEV_READ:
    nread = wip_read( c, buffer, sizeof( buffer) - 1);
    buffer[nread] = '\0';
    switch( *ctx) {
    case JUST_OPEN:
      TRACE( (1, "Received greeting from server \n") );
      wip_write( c, "NAME", strlen( "NAME"));
      *ctx = FORENAME_REQUEST_SENT;
      break;
    case FORENAME_REQUEST_SENT:
      TRACE( (1, "Forename \n" ) );
      wip_write( c, "FORENAME", strlen( "FORENAME"));
      *ctx = NAME_REQUEST_SENT;
      break;
    case NAME_REQUEST_SENT:
      TRACE( (1, "Name \n" ) );
      wip_write( c, "PHONE", strlen( "PHONE"));
      *ctx = PHONE_REQUEST_SENT;
```

```
      break;
    case PHONE_REQUEST_SENT:
      TRACE( (1, "Phone \n" ) );
      wip_write( c, "QUIT", strlen( "QUIT"));
      *ctx = QUIT_REQUEST_SENT;
      break;
    case QUIT_REQUEST_SENT:
      TRACE( (1, "Server says goodbye \n") );
      wip_close( c);
      break;
    }
  }
 case WIP_CEV_WRITE:
 case WIP_CEV_ERROR:
 case WIP_CEV_PEER_CLOSE:
   break;
}

void startClient() {
  static enum state state = JUST_OPEN;

  wip_channel_t client = wip_TCPClientCreate( SERVER_ADDRESS,
                                              SERVER_PORT,
                                              &commHandler,
                                              &state );
}
```

15.3. Advanced TCP Example

This is a complex example. It is a rudimentary chat server. Clients connect to the server, and first send an integer, known as their ID. If the client is the first one to send this ID, then it is put on hold until a second one sends the same ID (state WAIT_FOR_SECOND_CX). If it is the second one to send this ID, then it is connected to the first client with this ID. Once the two clients are connected, everything written by one client is forwarded to the dual client. If there are already two clients with this ID, any attempt by a third client to use the same ID is rejected (message EMSG_3RD_CONNECT).

```
/* How many connection can be handled simultaneously */
#define CX_NUM      16
/* Port number of the server */
#define SERVER_PORT 1235
/* Error messages */
#define EMSG_NO_CTX      "Error: no available context on server\n"
#define EMSG_3RD_CONNECT "Error: you're the 3rd to request that id\n"

/* Connection context */
struct {
  /* Number identifying the connection */
  s32 cx_id;
  enum {
    /* This context is currently unused */
    FREE,
    /* One connection has been made, waiting for the second */
    WAIT_FOR_SECOND_CX,
    /* Both clients are connected, they can chat together */
    CONNECTED
  } state;
  /* First client to connect */
  wip_channel_t cx0;
  /* Second client to connect */
  wip_channel_t cx1;
} cx_state;

/* Connection contexts pool */
static struct cx_state cx_table[CX_NUM];

/* Handling events on communication sockets */
void commHandler( wip_event_t *ev, struct cx_state *ctx) {
  s32 err;
  wip_channel_t c = ev->channel;
```

```
switch( ev->kind) {

case WIP_CEV_READ:
  /* Some data arrived, that can be read */
  if( NULL == ctx) {
    /* unconnected socket: read id */
    s32 i, id;
    /*wait for more data*/
    if( ev->content.read.readable < sizeof( id))
      return;
    wip_read( c, &id, sizeof( id));
    /* find any open cx with that id */
    for( i = 0; i < CX_NUM; i++) {
      if( cx_table[i].cx_id == id) {
        ctx = cx_table + i;
        switch( ctx->state) {

        case FREE:
          /* This entry is unused, its cx_id field is meaningless;
             continue to the next ctx. */
          break;

        case CONNECTED:
          /* Only two connections can use a given id */
          wip_write( c, EMSG_3RD_CONNECT, strlen( EMSG_3RD_CONNECT));
          wip_close( c);
          return;

        case WAITING_FOR_SECOND_CX:
          /* This is the 2nd connection with this id: complete the ctx,
             and register it with that channel */
          ctx->cx1 = c;
          ctx->cx_state = CONNECTED;
          wip_setCtx( c, ctx);
          return;
        }
      }
    }
    /* No connection found with this id; find a FREE ctx in the pool */
    for( i = 0; i < CX_NUM; i++) {
      if( FREE == cx_table[i].cx_state) {
        ctx = cx_table + i;
        wip_setCtx( c, ctx);
```

```
          ctx->cx0 = c;
          ctx->cx_state = WAITING_FOR_SECOND_CX;
          if( err < 0) goto error;
          return;
        }
      }

      /* No free cx context available in the pool */
      wip_write( c, NO_CTX_MSG, strlen( NO_CTX_MSG));
      wip_close( c);
      return;

    } else {
     /* [ev->kind == WIP_CEV_READ && ctx != NULL]: connection is already
established */
      void *buffer;



      wip_channel_t dual = (ctx->cx0 == c) ? ctx->cx1 : ctx->cx0;
      s32 writeable_on_dual;
      s32 readable = ev->content.read.readable;

      wip_getOpts( dual,
                   WIP_COPT_NWRITE, &writeable_on_dual,
                   WIP_COPT_END);
      if( writeable_on_dual < readable) return;
      buffer = adl_memGet( readable);
      wip_read( c, buffer, readable);
      wip_write( dual, buffer, readable);
      adl_memRelease( buffer);
      return;
    }

  case WIP_CEV_WRITE:
    /* There is some buffer space to write... Yet I've got nothing
       interesting to write in it: I'll write something when I'll receive
       something to read! */
    return;

  case WIP_CEV_ERROR:
  case WIP_CEV_PEER_CLOSE:
    /* If a socket closes, or something goes wrong, close the dual
       socket */
```

```
    if( ctx != NULL && ctx->cx_state == CONNECTED) {
      wip_close( ctx->cx0);
      wip_close( ctx->cx1);
      ctx->state = FREE;
    } else if( ctx != NULL) {
      wip_close( c);
      ctx->state = FREE;
    }
    else wip_close( c);
    return;
  }
}

/* Starting the server */
void initServer() {
  s32 i;
  wip_channel_t server;

  for( i = 0; i < CX_NUM; i++) cx_table[i].state = FREE;

  server = wip_TCPServerCreate( SERVER_PORT, commHandler, NULL);
}
```

15.4. Simple FTP Example

This program downloads a file named data.bin from the server ftp.sierrawireless.com and puts it in memory. However, since it makes no assumptions on the file's size, it requests it with wip_getFileSize() before allocating the buffer. Once the whole file has been read, the resulting buffer is passed to a DoSomethingWithIt() function.

For the sake of simplicity, this sample does no error checking.

```
#define SERVER "ftp.sierrawireless.com"
#define FILE_NAME "data.bin"
static u8 *buffer;
static int buf_size;

/* Handling events on the connection channel.*/
static evh_cx( wip_event_t *ev, void *ctx) {
  switch( ev->kind) {
    case WIP_CEV_OPEN:
      /* FTP connection just established*/
      wip_getFileSize( ev->channel, FILE_NAME);
      break;
    case WIP_CEV_DONE:
      /* response to the wip_getFileSize() call arrived. */
      buf_size = ev->content.done.aux;
      /* allocate the buffer */
      buffer = adl_getMem( buf_size);
      /* And start filling it with data */
      wip_getFile( ftp_cx, FILE_NAME, evh_data, NULL);
      break;
  }
     }
/* Handling events on the file transfer channel. */
static void evh_data( wip_event_t *ev, void *ctx) {
  static int nwritten;
  switch( ev->kind) {
   case WIP_CEV_OPEN:
     nwritten = 0;
     break;
case WIP_CEV_READ:
     nwritten += wip_read( ev->channel, buffer + nwritten,
                           buf_size - nwritten);
     /* We know that the whole file content is smaller than buf_size*/
     ASSERT( nwritten <= buf_size);
```

```
      break;

    case WIP_CEV_PEER_CLOSE:
      wip_close( ev->channel);
      DoSomethingWithIt( buffer, nwritten);
      break;
  }
}

/* When Internet Library is ready, open the FTP server */
void evh_bearer(wip_bearer_t b, s8 event, void *ctx) {
  if( WIP_BEV_IP_CONNECTED == event)
    wip_FTPCreate( SERVER, evh_cx, NULL);
}

int adl_main() {
  ...
  /* Configure a bearer. */
  wip_bearerOpen( ..., ..., evh_bearer, NULL);
  ...
    }
```

In a multithreaded environment, where blocking calls are acceptable, everything could have been put in a single thread, which would have been put asleep when waiting for events. The program would have looked like:

```
wip_blockingBearerStart( &bearer, …);
ftpcx = wip_blockingFTPCreate( SERVER);
size = wip_blockingGetFileSize( ftpcx, FILE_NAME);
buffer = adl_getMem( size);
nwritten = 0;
transfer = wip_blockingGetFile( ftpcx, FILE_NAME);
while( WIP_CSTATE_READY == wip_getState( transfer))
  nwritten += wip_blockingRead( transfer, buffer + nwritten,
                                size – nwritten);
wip_close( transfer);
doSomethingWithIt( buffer);
```

Notice that wip_blockingXxx() calls don’t exist in the current API; the snippet above is to be read as pseudo-code.

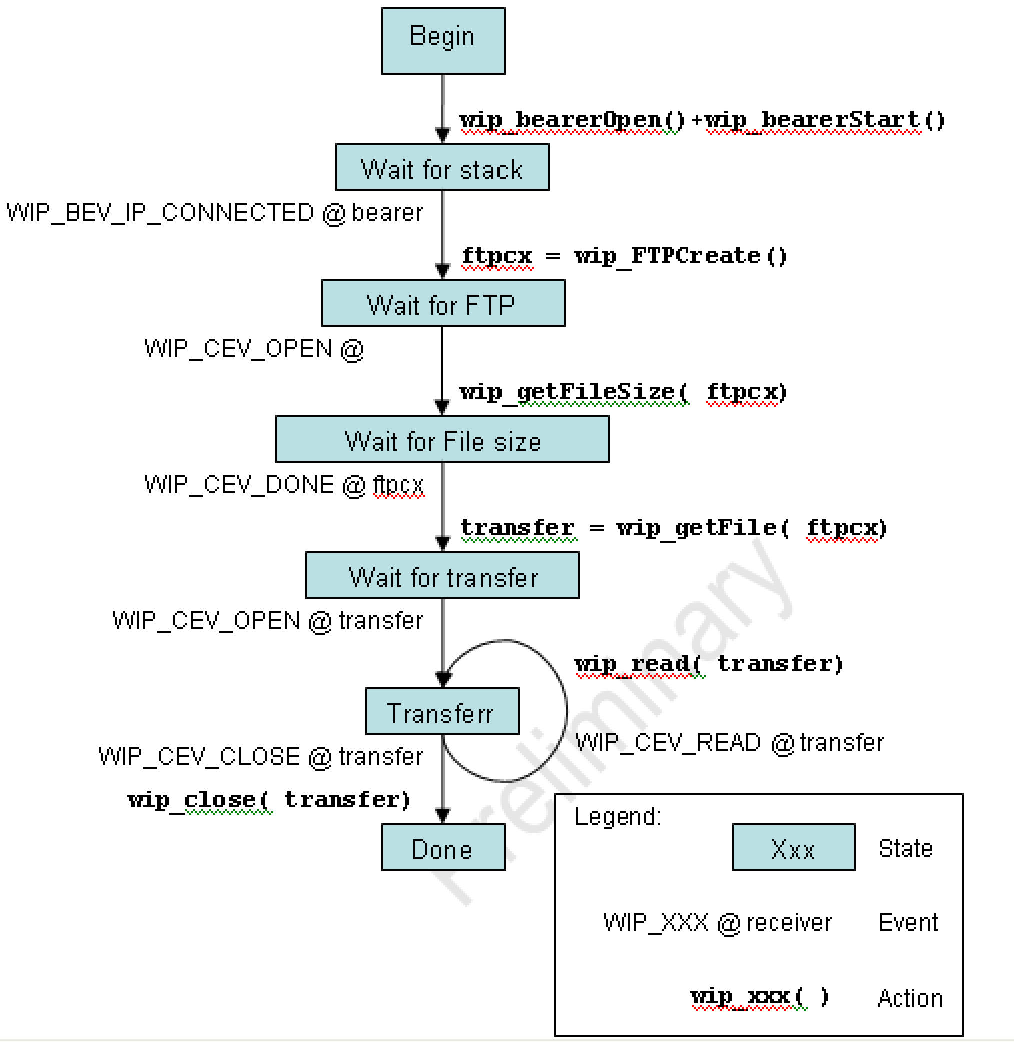


Figure 20. State machine of a simple FTP application

The corresponding state machine is represented above. It has the following noticeable property: each (event, receiver) couple occurs only once in the machine, which means there is no need to explicitly remember the machine's state: it can be deduced from the event. In a more complex example, it would be necessary to:

- create an enum type listing the possible state
- test the current state when an event happens
- update the state after an action is performed

In the event handlers, the switch statements would have looked like:

```
enum { STATE_YYY0, STATE_YYY1, /* etc. */ } state;

void evh_xxx( wip_event_t *ev, void *ctx) {
  switch( ev->kind) {
    case WIP_CEV_XXX0: switch( state) {
      case STATE_YYY0:
        /* Do whatever must be done when event XXX0 happens to ev->channel
           when in state YYY0 */
        someAction();
```

```
        /* relevant state transition. */
        state = STATE_YYY3;
        break;
      case STATE_YYY1:
        someOtherAction();
        state = STATE_YYY2;
        break;
        /* etc. */
    }

 case WIP_CEV_XXX1: switch( state) {
    /* etc. */
    }
    /* etc. */
  }
}
```

15.5. Advanced FTP Example

This program makes use of the file browsing API. It recursively downloads every files in an FTP server directory. As many downloads as possible are started concurrently; the program detects whenever TCP sockets are used (error WIP_CERR_RESOURCES).

```
TBD
```

15.6. Simple HTML Example

This example shows how to get a HTML page from a web server.

```
/* HTTP session */
wip_channel_t http;

/* event handler callback */
void http_event( wip_event_t *ev, void *ctx)
{
  wip_channel_t ch;
  s32 ret;
  ascii buf[256];

  /* get originating channel */
  ch = ev->channel;

  switch( ev->kind) {
  case WIP_CEV_OPEN:
    break;
  case WIP_CEV_READ:
    /* read html page */
    while( (ret = wip_read( ch, buf, sizeof( buf))) > 0) {
      /* ...store data... */
    }
    break;

  case WIP_CEV_PEER_CLOSE:
    /* html page has been received */
    /* get status code and reason string */
    wip_getOpts( ch,
                 WIP_COPT_HTTP_STATUS_CODE, &ret,
                 WIP_COPT_HTTP_REASON, buf, sizeof(buf),
                 WIP_COPT_END);
    if( ret != 200) {
       /* not OK… */
    }
    else
    {
      /* get type of document (should be 'text/html'...) */
      wip_getOpts( ch,
                   WIP_COPT_HTTP_HEADER, "content-type", buf, sizeof(buf),
```

```
                   WIP_COPT_END);
      if( wm_strcmp( buf, "text/html") == 0) {
         /* … */
      }
    }
    wip_close( ch);
    break;

  case WIP_CEV_ERROR:
    /* socket error... close channel */
    wip_close( ch);
    break;
  }
}

/* Application */
void MyFunction( void)
{
  /* Setup HTTP session */
  http = wip_HTTPClientCreateOpts(
                   NULL, NULL,
                   WIP_COPT_HTTP_HEADER,      "User-Agent",      "WIP-HTTP-
Client/1.0",
                   WIP_COPT_END);

  /* Get a HTML page */
  wip_getFileOpts ( http,
                    "http://www.sierrawireless.com",
                    http_event, NULL,
                    WIP_COPT_HTTP_HEADER, "Accept", "text/html",
                    WIP_COPT_END);
}
```

15.7. Generation of HTTP Header Example

This example shows how to generate a digest authorization header for HTTP session.

```
/* authentication parameters - see RFC 2617
ascii *www_user = "guest";
ascii *www_passwd = "123456";
ascii *www_cnonce = "abcdefgh";
u32 www_non_count = 0;
ascii *www_url = "http://myserver/private";

/* HTTP session */
wip_channel_t http;
ascii wwwauth[256];
ascii *www_our_auth;
wip_channel_t req_channel;   /* current request channel */

/* get authentication header sent by server */
  wip_getOpts( req_channel,
     WIP_COPT_HTTP_HEADER, "www-authenticate", wwwauth, sizeof(wwwauth),
     WIP_COPT_END);

/* check that server requested digest authentication */
if( wip_HTTPAuthScheme( wwwauth, "digest")) {
  /* compute authorization header */
  www_our_auth = wip_HTTPAuthDigest( wwwauth,
                                     www_url,
                                     WIP_HTTP_METHOD_GET,
                                     www_cnonce,
                                     www_non_count,
                                     www_user,
                                     www_passwd);
  /* close previous request */
  wip_close( req_channel);

  /* repeat previous request with our authentication header */
  req_channel = wip_getFileOpts( http,
                   www_url,
                   http_event, NULL,
```

```
                    WIP_COPT_HTTP_HEADER, "Accept", "text/html",
                    WIP_COPT_HTTP_HEADER, "Authorization", www_our_auth,
                    WIP_COPT_END);

  /* warning: dont forget to release www_our_auth buffer after
     request channel is closed by calling wip_HTTPAuthFree( www_our_auth);
*/
  }
}
```

15.8. Simple SMTP Example

This example shows how to send an Email using the SMTP client interface.

```
#include "adl_global.h"  // Global includes
#include "wip_smtp.h"    // Internet Library smtp services
/* Local variables */
/* SMTP defined strings set */
#define SMTP_CLIENT_TEST_PORT       25
const ascii * SMTP_STR_HOSTNAME = "192.168.1.5";
const ascii * SMTP_STR_USERNAME = "user01";
const ascii * SMTP_STR_PASSWORD = "user01";
const ascii * SMTP_STR_SENDERNAME = "OAT WIP sender";
const ascii * SMTP_STR_SENDER = "<user01@mydomain.fr>";
const ascii * SMTP_STR_REC = "<user02@mydomain.fr>,<user03@mydomain.fr>";
const ascii * SMTP_STR_CCREC = "";
const ascii * SMTP_STR_BCCREC = "<bcc01@mydomain.fr>,<bcc02@mydomain.fr>";
const ascii * SMTP_STR_SUBJ = "WIP sender mail test SMTP";


/* Pre formatted mail without header (Internet Library smtp generates the
header) */
const ascii * SMTP_STR_BODY = "SMTP predefined mail sample:
\r\n"
                              "
\r\n"
                              " This body message is intended to
\r\n"
                              " illustrate the SMTP client interfaces
\r\n"
                              " used for a pre configured mail delivery
\r\n"
                              " with the predefined recipients lists.
\r\n"
                              "
\r\n"
                              "
\r\n"
                              " The WIPSender . . .
\r\n"
                              "
\r\n"


/* Pre formatted mail + header with file attachment (appli pre formats the
  header) */
```

```
const ascii *SMTP_STR_ATT_MAIL =
   "Subject: Appli preformatted mail with attachment \r\n"
   "From: appli name sender <wiptester@yahoo.fr>     \r\n"
   "To: receiver@domain.fr \r\n"
   "Bcc: receiver@domain.fr \r\n"
   "MIME-Version: 1.0\r\n"
   "Content-Type: Multipart/Mixed; \r\n"
   "    boundary=\"----=__marking_attachment\"\r\n"
   "\r\n"
   "\r\n" // This end the appli pre formatted header
   "This is a multi-part message in MIME format.\r\n"
   /* Start of message text section */
   "------=__marking_attachment\r\n"
   "Content-Type: text/plain\r\n"
   "charset=\"us-ascii\"\r\n"
   "Content-Transfer-Encoding: quoted-printable\r\n"
   "Hello, this is a demo mail with an attachement file\r\n"
   "\r\n"
   "\r\n"
   /* Start of attachment section */
   "------=__marking_attachment\r\n"
   "         name=\"attached_file.txt\"\r\n"
   "Content-Transfer-Encoding: base64\r\n"
   "Content-Description: Attached txt file encrypted mime64\r\n"
   "Content-Disposition: attachment; filename=\"attached_file.txt\" \r\n"
   "\r\n"
   /* Start of encrypted attachment block */

"VGhpcyBib2R5IG1lc3NhZ2UgaXMgaW50ZW5kZWQgdG8gaWxsdXN0cmF0ZSB0aGUgDQpTTVRQIE
Ns

aWVudCBJbnRlcmZhY2VzIHVzZWQgZm9yIGEgICAgICAgICAgICAgICANCnByZSBjb25maWd1cmV
kI

G1haWwgZGVsaXZlcnkgd2l0aCB0aGUgcHJlZGVmaW5lZA0KcmVjaXBpZW50cyBsaXN0cy4NCiAg
IC

AgICAgICAgICAgDQogICAgICAgICAgICAgICAgIA0KVGhlIFdpcFNlbmRlci4uLiANCiAgICAgI
CA

gICAgICAgICAgDQo= \r\n\r\n\r\n"
/* End of attachment section */
"------=__marking_attachment--\r\n";


typedef struct
```

```
{
  wip_channel_t CnxChannel;            // session channel
  wip_channel_t DataChannel;           // data channel
  u8            *pdataBuffer;          // mail data pointer
  u32            dataLength;           // mail data length
} smtp_ClientTestCtx_t;

smtp_ClientTestCtx_t smtp_ClientTestCtx;

/* Constants */
static s32 smtp_ClientTestPutFile(void);

/* Local functions */
/* Handler for the SMTP connection channel */
static void smtp_ClientTestCnxHandler(wip_event_t *ev, void *ctx)
{
  smtp_ClientTestCtx_t *pSmtpClientCtx =
    (smtp_ClientTestCtx_t *)&smtp_ClientTestCtx;
  u32 StatusCode = 0;
  u32 ErrCode = 0;
  ascii **pErrorString;
  u32 ret = 0;
  TRACE(( 4, "smtp_ClientTestCnxHandler: %d 0x%x\n", ev->kind, ctx ));
  switch(ev->kind)
  {
    case WIP_CEV_OPEN:
      TRACE(( 4, "smtp_ClientTestCnxHandler: WIP_CEV_OPEN\n" ));
      // open the DATA channel
      smtp_ClientTestPutFile();
      break;
   case WIP_CEV_PEER_CLOSE:
      TRACE(( 4, "smtp_ClientTestCnxHandler: WIP_CEV_PEER_CLOSE\n" ));
      // Close DATA and SESSION channels
      if (pSmtpClientCtx->CnxChannel != NULL)
      {
        wip_close(pSmtpClientCtx->CnxChannel);
        pSmtpClientCtx->CnxChannel = NULL;
      }
```

```
      break;
    case WIP_CEV_ERROR:
      ret = wip_getOpts( pSmtpClientCtx->CnxChannel,
                         WIP_COPT_ERROR, &ErrCode,
                         WIP_COPT_END);
      // Get the last SMTP protocol Status Code
      ret = wip_getOpts( pSmtpClientCtx->CnxChannel,
              WIP_COPT_SMTP_STATUS_CODE, &StatusCode, &pErrorString,
              WIP_COPT_END);
      TRACE((   4,   "smtp_ClientTestCnxHandler:   WIP_CEV_ERROR   %d   \n",
ErrCode));
      // Close CNX channel
      if (pSmtpClientCtx->CnxChannel != NULL)
      {
        wip_close(pSmtpClientCtx->CnxChannel);
        pSmtpClientCtx->CnxChannel = NULL;
      }
      break;
    default:
      break;
  }
}

/* Handler for the SMTP data channel WIP_CEV_WRITE event */
static void smtp_ClientTestDataWriteHandler(void)
{
  smtp_ClientTestCtx_t *pSmtpClientCtx =
   (smtp_ClientTestCtx_t *)&smtp_ClientTestCtx;
  // While there are DATA to send
  while( pSmtpClientCtx->dataLength > 0 )
  {
    int WrittenBytes;
    WrittenBytes = wip_write( pSmtpClientCtx->DataChannel,
                              pSmtpClientCtx->pdataBuffer,
                              pSmtpClientCtx->dataLength );
    if (WrittenBytes <= 0)
    {
      // If wip_write() not ready or failed, end loop
```

```
      break;
    }
    // Update current pointer and length
    pSmtpClientCtx->pdataBuffer += WrittenBytes;
    pSmtpClientCtx->dataLength -= WrittenBytes;
  }
  // Check if entire data block has been written
  if (pSmtpClientCtx->dataLength == 0)
  {
    // Close DATA channel
    TRACE(( 4, "smtp_ClientTest: Close DATA channel\n" ));
    if (pSmtpClientCtx->DataChannel != NULL)
    {
      wip_close(pSmtpClientCtx->DataChannel);
      pSmtpClientCtx->DataChannel = NULL;
    }
    TRACE(( 4, "smtp_ClientTest: Close SESSION channel\n" ));
    // Close CNX channel
    if (pSmtpClientCtx->CnxChannel != NULL)
    {
      wip_close(pSmtpClientCtx->CnxChannel);
      pSmtpClientCtx->CnxChannel = NULL;
    }
  }
}

/* Handler for the SMTP data channel */
static void smtp_ClientTestDataHandler(wip_event_t *ev, void *ctx)
{
  smtp_ClientTestCtx_t *pSmtpClientCtx =
   (smtp_ClientTestCtx_t *)&smtp_ClientTestCtx;
  u32 Index, Length;
  s32 ret = 0;
  ascii **pErrorString;
  u32 ErrorCode;
  u32 StatusCode;
  TRACE(( 4, "smtp_ClientTestDataHandler: %d 0x%x\n", ev->kind, ctx ));
  switch(ev->kind)
```

```
  {
    case WIP_CEV_OPEN:
      TRACE(( 4, "smtp_ClientTestDataHandler: WIP_CEV_OPEN\n" ));
      // Send a mail with attachment file
      pSmtpClientCtx->pdataBuffer = SMTP_STR_ATT_MAIL;
      pSmtpClientCtx->dataLength  = wm_strlen(SMTP_STR_ATT_MAIL);
      break;
    case WIP_CEV_PEER_CLOSE:
      TRACE(( 4, "smtp_ClientTestDataHandler: WIP_CEV_PEER_CLOSE\n" ));
      break;
    case WIP_CEV_ERROR:
      ret = wip_getOpts(pSmtpClientCtx->DataChannel,
                        WIP_COPT_ERROR, &ErrorCode,
                        WIP_COPT_END);
      ret = wip_getOpts(pSmtpClientCtx->DataChannel,
              WIP_COPT_SMTP_STATUS_CODE, &StatusCode, &pErrorString,
              WIP_COPT_END);
      TRACE(( 4, "smtp_ClientTestDataHandler: WIP_CEV_ERROR %d\n",
              ErrorCode));
      break;
    case WIP_CEV_WRITE:
      // If opened and write ready, directly send mail through DATA channel
      smtp_ClientTestDataWriteHandler();
      break;
    default:
      break;
  }
}

static s32 smtp_ClientTestCreate(void)
{
  smtp_ClientTestCtx_t *pSmtpClientCtx =
   (smtp_ClientTestCtx_t *)&smtp_ClientTestCtx;
  wip_channel_t CnxChannel;
  wip_channel_t DataChannel;
  s32 ret = 0;
  // Session/Connection creation
  CnxChannel = wip_SMTPClientCreateOpts(
```

```
                  SMTP_STR_HOSTNAME,         // hostname
                  smtp_ClientTestCnxHandler, // handler fct
                  NULL,                      // ctx
                  // Optional parameters (mandatory at creation)
                  WIP_COPT_PEER_PORT, SMTP_CLIENT_TEST_PORT,
                  // Optional
                  // (if the both field are not specified, there
                  // is no AUTH LOGIN sequence)
                  WIP_COPT_USER,       SMTP_STR_USERNAME,
                  WIP_COPT_PASSWORD,   SMTP_STR_PASSWORD,
                  WIP_COPT_SMTP_AUTH_TYPE, WIP_SMTP_AUTH_MIME64,
                  WIP_COPT_END);
  if (CnxChannel == NULL)
  {
    TRACE(( 1, "cannot create smtp session channel\n" ));
    return(-1);
  }
  else
  {
    u32 port, authtype;
    ascii **username, **password, **hostname;
    // Get the specified options
    ret = wip_getOpts( CnxChannel,
                        WIP_COPT_ADDR, &hostname,
                        WIP_COPT_USER, &username,
                        WIP_COPT_PASSWORD, &password,
                        WIP_COPT_PEER_PORT, &port,
                        WIP_COPT_SMTP_AUTH_TYPE, &authtype,
                        WIP_COPT_END);
    smtp_ClientTestCtx.CnxChannel  = CnxChannel;
  }
  return(ret);
}

static s32 smtp_ClientTestPutFile(void)
{
  smtp_ClientTestCtx_t *pSmtpClientCtx =
   (smtp_ClientTestCtx_t *)&smtp_ClientTestCtx;
```

```
wip_channel_t CnxChannel = (wip_channel_t)smtp_ClientTestCtx.CnxChannel;
wip_channel_t DataChannel;
s32 ret = 0;
TRACE(( 4, "smtp_ClientTestPutFile\n" ));
// Data channel creation
// Application generates the mail header for attachments
DataChannel = wip_putFileOpts(
                CnxChannel,                  // Session channel
                NULL,                        // string
                smtp_ClientTestDataHandler, // handler fct
                NULL,                        // ctx
               // (MANDATORY for channel creation)
               WIP_COPT_SMTP_SENDERNAME, SMTP_STR_SENDERNAME,
               WIP_COPT_SMTP_SENDER,     SMTP_STR_SENDER,
               WIP_COPT_SMTP_REC,        SMTP_STR_REC,
               WIP_COPT_SMTP_CC_REC,     SMTP_STR_CCREC,
               WIP_COPT_SMTP_BCC_REC,    SMTP_STR_BCCREC,
               WIP_COPT_SMTP_SUBJ,       SMTP_STR_SUBJ,
               // This option indicates if wip smtp should generate
               // the mail header or if appli is in charge of it
               // (for mail attachment for example)
               WIP_COPT_SMTP_FORMAT_HEADER, 0, // default case
               WIP_COPT_END);
if (DataChannel == NULL)
{
  TRACE(( 1, "cannot create smtp data channel\n" ));
  return(-1);
}
else
{
  ascii **sendername, **sender, **rec, **cc_rec, **bcc_rec, **subject;
  u32 FormatHeader;
  // Get the specified options
  ret = wip_getOpts( DataChannel,
                     WIP_COPT_SMTP_SENDERNAME,    &sendername,
                     WIP_COPT_SMTP_SENDER,        &sender,
                     WIP_COPT_SMTP_REC,           &rec,
```

```
                        WIP_COPT_SMTP_CC_REC,         &cc_rec,
                        WIP_COPT_SMTP_BCC_REC,        &bcc_rec,
                        WIP_COPT_SMTP_SUBJ,           &subject,
                        WIP_COPT_SMTP_FORMAT_HEADER, &FormatHeader,
                        WIP_COPT_END);
    smtp_ClientTestCtx.DataChannel = DataChannel;
    // From this point a WIP_CEV_WRITE will be notify,
    // to signal to the application that it can start wip_write()
  }
  return(ret);
}

/* Called once the Internet Library IP stack is fully initialized.
   This is the starting point of user applications. */
void appli_entry_point(void)
{
  TRACE (( 1, "SMTP Client Service test application : Init" ));
  smtp_ClientTestCreate();
}
```

15.9. Simple POP3 Example

This example shows how to retrieve an Email using the POP3 client inerface.

```
#include "adl_global.h"  // Global includes
#include "wip_pop3.h"    // POP3 services
#include "wip_file.h"    // wip_fileInfo_t


/* Local variables */
/* Constants */
#define POP3_CLIENT_TEST_PORT 110
// POP3 defined strings set
const ascii * POP3_STR_HOSTNAME   = "192.168.1.5";
const ascii * POP3_STR_USERNAME   = "user02";
const ascii * POP3_STR_PASSWORD   = "user02";


typedef struct
{
  wip_channel_t CnxChannel;           // session channel
  wip_channel_t ListChannel;          // list channel
  wip_channel_t DataChannel;          // data channel
  u32            currentPort;         // current WuP module port
  // List buffer
  u8            *plistdataAllocBuffer; // mail list data pointer allocated
  u8            *plistdataBuffer;      // mail list data pointer
  u32            listdataLength;       // mail list data length
  u32            listdataLengthMax;    // mail list data max length
  // Mail content buffer
  u8            *pdataAllocBuffer;     // mail data pointer allocated
  u8            *pdataBuffer;          // mail data pointer
  u32            dataLength;           // mail data length
  u32            dataLengthMax;        // mail list data max length
  // Last mail context
  u32            totalMails;
  u32            deletedId;
  u32            retrievedId;
  u32            headerLines;
  u32            totalMailSize;
```

```
  u32           mailSize;
  // FCM context
  s8            v24Handle;
} pop3_ClientTestCtx_t;
pop3_ClientTestCtx_t pop3_ClientTestCtx = {0};

/* Constants */
static s32 pop3_ClientTestListOpts(void);
static s32 pop3_ClientTestGetFile(u32 MailId);

/* Local functions */

/* Handler for the POP3 connection channel  */
static void pop3_ClientTestCnxHandler(wip_event_t *ev, void *ctx)
{
  pop3_ClientTestCtx_t *pPop3ClientCtx =
   (pop3_ClientTestCtx_t *)&pop3_ClientTestCtx;
  s32 ret = 0;
  ascii *pString;
  ascii **pErrorString;
  u32 ErrCode;
  u32 StatusCode;
  TRACE(( 4, "pop3_ClientTestCnxHandler: %d 0x%x\n", ev->kind, ctx ));
  switch(ev->kind)
  {
    case WIP_CEV_OPEN:
      TRACE(( 4, "pop3_ClientTestCnxHandler: WIP_CEV_OPEN\n" ));
      // Open a listOpts channel
      pop3_ClientTestListOpts();
      break;
    case WIP_CEV_PEER_CLOSE:
      TRACE(( 4, "pop3_ClientTestCnxHandler: WIP_CEV_PEER_CLOSE\n" ));
      if (pPop3ClientCtx->CnxChannel != NULL)
      {
        wip_close(pPop3ClientCtx->CnxChannel);
        pPop3ClientCtx->CnxChannel = NULL;
      }
      break;
    case WIP_CEV_ERROR:
      ret = wip_getOpts(pPop3ClientCtx->CnxChannel,
```

```
                        WIP_COPT_ERROR, &ErrCode,
                        WIP_COPT_END);
      ret = wip_getOpts(pPop3ClientCtx->CnxChannel,
              WIP_COPT_POP3_STATUS_CODE, &StatusCode, &pErrorString,
              WIP_COPT_END);
      TRACE((  4,  "pop3_ClientTestCnxHandler:  WIP_CEV_ERROR  %d\n",  ErrCode
));
      if (pPop3ClientCtx->CnxChannel != NULL)
      {
        wip_close(pPop3ClientCtx->CnxChannel);
        pPop3ClientCtx->CnxChannel = NULL;
      }
      break;
   case WIP_CEV_DONE:
      TRACE(( 4, "pop3_ClientTestCnxHandler: WIP_CEV_DONE\n" ));
      break;
    default:
      break;
  }
}


/* Handler for the POP3 list channel WIP_CEV_READ event */
static void pop3_ClientTestListReadHandler(void)
{
  pop3_ClientTestCtx_t *pPop3ClientCtx =
    (pop3_ClientTestCtx_t *)&pop3_ClientTestCtx;
  // While there are DATA to read
  while( pPop3ClientCtx->listdataLengthMax > 0 )
  {
    int ReadBytes;
    ReadBytes = wip_read( pPop3ClientCtx->ListChannel,
                          pPop3ClientCtx->plistdataBuffer,
                          pPop3ClientCtx->listdataLengthMax );
    TRACE(( 4, "pop3_ClientTestListReadHandler: read %d/%d \n",
            ReadBytes, pPop3ClientCtx->listdataLength));
    if (ReadBytes <= 0)
    {
      // If wip_read() not ready or failed, end loop
      break;
```

```
    }
    // Update current pointer and length
    pPop3ClientCtx->plistdataBuffer    += ReadBytes;
    pPop3ClientCtx->listdataLengthMax  -= ReadBytes;
    pPop3ClientCtx->listdataLength     += ReadBytes;
  }
}
/* Handler for the POP3 list channel */
static void pop3_ClientTestListHandler(wip_event_t *ev, void *ctx)
{
  pop3_ClientTestCtx_t *pPop3ClientCtx =
   (pop3_ClientTestCtx_t *)&pop3_ClientTestCtx;
  u32 ret, mails, mailsize;
  switch(ev->kind)
  {
    case WIP_CEV_OPEN:
      TRACE(( 4, "pop3_ClientTestListHandler: WIP_CEV_OPEN\n" ));
      // Get total mail and size
      // Get the specified options
      ret = wip_getOpts( pPop3ClientCtx->CnxChannel,
                         WIP_COPT_POP3_NB_MAILS, &mails,
                         WIP_COPT_POP3_MAILSIZE, &mailsize,
                         WIP_COPT_END);
      pPop3ClientCtx->totalMailSize = mailsize;
      pPop3ClientCtx->totalMails    = mails;
      pPop3ClientCtx->plistdataAllocBuffer =
         (u8 *)adl_memGet( mails * WIP_POP3_FILEINFO_SIZE );
      if (pPop3ClientCtx->plistdataAllocBuffer != NULL)
      {
        pPop3ClientCtx->plistdataBuffer =
          pPop3ClientCtx->plistdataAllocBuffer;
        pPop3ClientCtx->listdataLengthMax = mails * WIP_POP3_FILEINFO_SIZE;
        pPop3ClientCtx->listdataLength    = 0;
      }
      break;
    case WIP_CEV_PEER_CLOSE:
      break;
    case WIP_CEV_ERROR:
      break;
    case WIP_CEV_READ:
```

```
      pop3_ClientTestListReadHandler();
      break;
    case WIP_CEV_DONE:
      {
        u32 i;
        wip_fileInfo_t *pwip_fileInfo_tmp = NULL;
        u32 MailId   = 0;
        u32 MailSize = 0;
        // The entire list has been read
        pwip_fileInfo_tmp =
           (wip_fileInfo_t *)pPop3ClientCtx->plistdataAllocBuffer;
       TRACE((   4,   "pop3_ClientTestListHandler:   WIP_CEV_DONE   0x%x   %d
bytes\n",
                pwip_fileInfo_tmp, pPop3ClientCtx->listdataLength));
        // Dump the received list Internet Library info structure
        i = 0;
        while(i < pPop3ClientCtx->listdataLength)
        {
          MailId =
            wm_atoi( pwip_fileInfo_tmp->entries[WIP_FOPT_NAME].ascii );
          MailSize = pwip_fileInfo_tmp->entries[WIP_FOPT_SIZE].u32;
          i += WIP_POP3_FILEINFO_SIZE;
          pwip_fileInfo_tmp  =
            (wip_fileInfo_t *)(pPop3ClientCtx->plistdataAllocBuffer+i);
        }
        if (pPop3ClientCtx->plistdataAllocBuffer != NULL)
        {
          adl_memRelease(pPop3ClientCtx->plistdataAllocBuffer);
          pPop3ClientCtx->plistdataAllocBuffer = NULL;
        }
        if (pPop3ClientCtx->ListChannel != NULL)
        {
          wip_close(pPop3ClientCtx->ListChannel);
          pPop3ClientCtx->ListChannel = NULL;
        }
        // If at least one mail
        if (pPop3ClientCtx->totalMails != 0)
        {
          pop3_ClientTestGetFile(1);
```

```
        }
      }
      break;
    default:
      break;
  }
}

/* Handler for the POP3 data channel WIP_CEV_READ event */
static void pop3_ClientTestDataReadHandler(void)
{
  pop3_ClientTestCtx_t *pPop3ClientCtx =
   (pop3_ClientTestCtx_t *)&pop3_ClientTestCtx;
  // While there are DATA to send
  while(pPop3ClientCtx->dataLengthMax > 0)
  {
    int ReadBytes;
    ReadBytes = wip_read( pPop3ClientCtx->DataChannel,
                          pPop3ClientCtx->pdataBuffer,
                          pPop3ClientCtx->dataLengthMax );
    TRACE(( 4, "pop3_ClientTestReadHandler: read %d/%d \n",
            ReadBytes, pPop3ClientCtx->dataLength));
    if (ReadBytes <= 0)
    {
      // If wip_read() not ready or failed, end loop
      break;
    }
    // Update current pointer and length
    pPop3ClientCtx->pdataBuffer    += ReadBytes;
    if (pPop3ClientCtx->dataLengthMax >= ReadBytes)
    {
      pPop3ClientCtx->dataLengthMax  -= ReadBytes;
    }
    else
    {
      pPop3ClientCtx->dataLengthMax = 0;
    }
    pPop3ClientCtx->dataLength     += ReadBytes;
  }
```

```
}

/* Timer handler  */
static void pop3_ClientTestTimerHandler ( u8 ID )
{
  pop3_ClientTestCtx_t *pPop3ClientCtx =
   (pop3_ClientTestCtx_t *)&pop3_ClientTestCtx;
  // Release alloc buffer
  TRACE(( 4, "pop3_ClientTestTimerHandler: release mail buffer\n"));
  if (pPop3ClientCtx->pdataAllocBuffer != NULL)
  {
    adl_memRelease(pPop3ClientCtx->pdataAllocBuffer);
    pPop3ClientCtx->pdataAllocBuffer = NULL;
  }
  // Close data channel
  TRACE(( 4, "pop3_ClientTestTimerHandler: Close data channel\n"));
  if (pPop3ClientCtx->DataChannel != NULL)
  {
    wip_close(pPop3ClientCtx->DataChannel);
    pPop3ClientCtx->DataChannel = NULL;
  }
  // Close session channel
  TRACE(( 4, "pop3_ClientTestTimerHandler: Close session channel\n"));
  if (pPop3ClientCtx->CnxChannel != NULL)
  {
    wip_close(pPop3ClientCtx->CnxChannel);
  }
}

/* Handler for the POP3 data channel */
ascii MailRspStr [ 550 ];
static void pop3_ClientTestDataHandler(wip_event_t *ev, void *ctx)
{
  pop3_ClientTestCtx_t *pPop3ClientCtx =
   (pop3_ClientTestCtx_t *)&pop3_ClientTestCtx;
  u32 Index, Max;
  s32 ret = 0;
  u32 mailsize = 0;
  TRACE(( 4, "pop3_ClientTestDataHandler: %d 0x%x\n", ev->kind, ctx ));
```

```
  switch(ev->kind)
  {
    case WIP_CEV_OPEN:
      TRACE(( 4, "pop3_ClientTestDataHandler: WIP_CEV_OPEN\n"));
      // Get the specified options
      ret = wip_getOpts( pPop3ClientCtx->DataChannel,
                         WIP_COPT_POP3_MAILSIZE, &mailsize,
                         WIP_COPT_END);
      TRACE((   4,   "pop3_ClientTestDataHandler:   mailsize   =   %d   \n",
mailsize));
      if (mailsize != 0)
      {
        pPop3ClientCtx->pdataAllocBuffer =
          (u8 *)(u8 *)adl_memGet( mailsize );
        if (pPop3ClientCtx->pdataAllocBuffer != NULL)
        {
          TRACE(( 4, "pop3_ClientTestDataHandler: alloc 0x%x[%d]\n",
                     pPop3ClientCtx->pdataAllocBuffer, mailsize));
          pPop3ClientCtx->pdataBuffer = pPop3ClientCtx->pdataAllocBuffer;
          pPop3ClientCtx->dataLengthMax = mailsize;
          pPop3ClientCtx->dataLength    = 0;
        }
      }
      else
      {
        TRACE(( 4,
          "pop3_ClientTestDataHandler: Error mailsize null!!!! \n"));
        wip_close(pPop3ClientCtx->DataChannel);
      }
      break;
    case WIP_CEV_PEER_CLOSE:
      break;
    case WIP_CEV_ERROR:
      break;
    case WIP_CEV_READ:
      pop3_ClientTestDataReadHandler();
      break;
    case WIP_CEV_DONE:
      // The entire mail has been read
      TRACE((   4,   "pop3_ClientTestDataHandler:   WIP_CEV_DONE   0x%x   %d
bytes\n",
```

```
              pPop3ClientCtx->pdataAllocBuffer,             pPop3ClientCtx-
>dataLength));
    // Tempo to wait for DATA channel properly closing
    // Set 2s timer
    // before closing session channel
    adl_tmrSubscribe ( FALSE, 20, ADL_TMR_TYPE_100MS,
                       pop3_ClientTestTimerHandler );
      break;
    default:
      break;
  }
}

/* POP3 demo send the mail */
static s32 pop3_ClientTestCreate(void)
{
  pop3_ClientTestCtx_t *pPop3ClientCtx =
    (pop3_ClientTestCtx_t *)&pop3_ClientTestCtx;
  wip_channel_t CnxChannel;
  wip_channel_t DataChannel;
  s32 ret = 0;
  // Session/Connection creation
  CnxChannel = wip_POP3ClientCreateOpts(
                 (ascii *)POP3_STR_HOSTNAME,// hostname
                  pop3_ClientTestCnxHandler,// handler fct
                  NULL,                     // ctx
                  // Optional parameters (mandatory at creation)
                  WIP_COPT_PEER_PORT, POP3_CLIENT_TEST_PORT,
                  WIP_COPT_USER,      POP3_STR_USERNAME,
                  WIP_COPT_PASSWORD,  POP3_STR_PASSWORD,
                  WIP_COPT_END);
  if (CnxChannel == NULL)
  {
    TRACE(( 1, "cannot create pop3 session channel\n" ));
    return(-1);
  }
  else
  {
    u32 port;
    ascii **username, **password, **hostname;
```

```
    // Get the specified options
    ret = wip_getOpts( CnxChannel,
                       WIP_COPT_ADDR,            &hostname,
                       WIP_COPT_USER,            &username,
                       WIP_COPT_PASSWORD,        &password,
                       WIP_COPT_PEER_PORT,       &port,
                       WIP_COPT_END);
   pop3_ClientTestCtx.CnxChannel  = CnxChannel;
  }
  return(ret);
}


/* POP3 demo send the mail */
static s32 pop3_ClientTestListOpts(void)
{
  wip_channel_t CnxChannel = (wip_channel_t)pop3_ClientTestCtx.CnxChannel;
  wip_channel_t ListChannel;
  s32 ret = 0;
  // List channel creation
  ListChannel = wip_listOpts(CnxChannel,                // Session channel
                             NULL,                        // string
                             pop3_ClientTestListHandler, // handler fct
                             NULL);                       // ctx
  if (ListChannel == NULL)
  {
    TRACE(( 1, "cannot create pop3 list channel\n" ));
    return(-1);
  }
  pop3_ClientTestCtx.ListChannel = ListChannel;
  // From this point a WIP_CEV_READ will be notify,
  // to signal to the application that it can start wip_read()
  // to get the mail listing and sizes
  return(ret);
}


static s32 pop3_ClientTestGetFile(u32 MailId)
{
  wip_channel_t CnxChannel = (wip_channel_t)pop3_ClientTestCtx.CnxChannel;
  wip_channel_t DataChannel;
```

```
  s32 ret = 0;
  ascii MailIdStr[10];
  wm_sprintf(&MailIdStr[0], "%d", MailId);
  TRACE(( 4, "pop3_ClientTestGetFile: %d %d\n", MailId, strlen(MailIdStr)
));
  // Data channel creation
  DataChannel = wip_getFileOpts(CnxChannel,    // Session channel
                              &MailIdStr[0], // string

                              pop3_ClientTestDataHandler, // handler fct
                              NULL //ctx
                              );
  if (DataChannel == NULL)
  {
    TRACE(( 1, "cannot create pop3 data channel\n" ));
    return(-1);
  }
  pop3_ClientTestCtx.DataChannel = DataChannel;
  // From this point a WIP_CEV_WRITE will be notify,
  // to signal to the application that it can start wip_write()
  return(ret);
}

static s32 pop3_ClientTestDeleteFile(u32 MailId)
{
  ascii MailIdStr[10];
  wip_channel_t CnxChannel = (wip_channel_t)pop3_ClientTestCtx.CnxChannel;
  wm_sprintf(&MailIdStr[0], "%d", MailId);
   TRACE(( 4, "pop3_ClientTestDeleteFile: %d %d\n",
           MailId, strlen(MailIdStr) ));
  wip_deleteFile(CnxChannel, &MailIdStr[0]);
}
/* Customer application initialization */
void appli_entry_point ( adl_InitType_e InitType )
{
  TRACE (( 1, "POP3 Client Service test application : Main" ));
  pop3_ClientTestCreate();
}
```

15.10. Simple MMS Example

This example shows how to send a MMS using the MMS client interface.

```
#include "adl_global.h"  /* Global includes */
#include "wip.h"
#include "wip_mms.h"   /* MMS services */

/* Global structures */
wip_channel_t HTTPCnxChannel;        /* HTTP session channel */
wip_mms_t p_mmsCtrl;                /* MMS control structure */
/* Buffer for the image and sound */
static const u8 INTRUDER_ALERT[] = {
  0x23, 0x21, 0x41, 0x4D, 0x52, 0x0A, 0x3C, 0x07, 0x0E, 0x9B, 0xB0, 0x36,
  0x2A, 0x44, 0x6C, 0xEE, 0xE5, 0xBF, 0x27, 0x77, 0x76, 0x44, 0xC0, 0x00,
  0x67, 0x48, 0x25, 0x88, 0xAC, 0x08, 0x00, 0x00, 0x5C, 0x5A, 0xC4, 0x56,
  0x09, 0x30, 0x3C,
   …
   …
  0x09, 0x0A, 0x9A, 0xB4, 0xA2, 0x6E, 0x09, 0x5E, 0x17, 0xE9, 0x68, 0xD4,
  0x81, 0xB7, 0xD6, 0x26, 0xB6, 0x5F, 0x72, 0x07, 0xD3, 0x2B, 0x85, 0xAC,
  0x78, 0x88, 0xDF, 0x9D, 0x80, 0x9F, 0xF0
};

static unsigned char image[] = {
  0xff, 0xd8, 0xff, 0xe0, 0x00, 0x10, 0x4a, 0x46, 0x49, 0x46, 0x00, 0x01,
  0x01, 0x01, 0x00, 0x60, 0x00, 0x60, 0x00, 0x00, 0xff, 0xdb, 0x00, 0x43,
  0x00, 0x08, 0x06, 0x06, 0x07, 0x06, 0x05, 0x08, 0x07, 0x07, 0x07, 0x09,
  0x09, 0x08, 0x0a,
  …
  …
  0xa8, 0x02, 0xd2, 0x9a, 0x95, 0x4d, 0x56, 0x56, 0xa9, 0x54, 0xd0, 0x05,
  0x85, 0x35, 0x20, 0x6a, 0xae, 0xad, 0x52, 0x03, 0x40, 0x13, 0x66, 0x97,
  0x35,0x10, 0x6a, 0x5d, 0xd4, 0x01, 0x2e, 0xea, 0x37, 0x54, 0x7b, 0xa8,
  0xdd, 0x40, 0x0f, 0x26, 0x9a, 0x4d, 0x37, 0x75, 0x34, 0xb5, 0x00, 0x7f,
  0xff, 0xd9, 0xd9
};

#define NAME   "test.jpg"
#define MMS_SERVER_IP   "10.151.0.1"
#define MMS_SERVER_PORT   8080

/* Function prototype */
```

```
static void appli_entry_point();
static void http_ClientTestDataHandler( wip_event_t *ev, void *ctx);

static void statuscallback(wip_mms_t mms, u32 status, void * ctx)
{
  TRACE (( 1,"statuscallback"));
  switch(status)
  {
    case WIP_MMS_STATUS_OK:
      TRACE (( 1,"MMS sent status : OK"));
    break;
    case WIP_MMS_STATUS_SERVICE_DENIED:
      TRACE (( 1,"MMS sent status : Service denied"));
    break;
    case WIP_MMS_STATUS_MESSAGE_FORMAT_CORRUPT:
      TRACE (( 1,"MMS sent status : Message format corrupt"));
    break;
    case WIP_MMS_STATUS_SENDING_ADDRESS_UNRESOLVED:
      TRACE (( 1,"MMS sent status : Sending address unresolved"));
    break;
    case WIP_MMS_STATUS_MESSAGE_NOT_FOUND:
      TRACE (( 1,"MMS sent status : Message not found"));
    break;
    case WIP_MMS_STATUS_NETWORK_PROBLEM:
      TRACE (( 1,"MMS sent status : Network problem"));
    break;
    case WIP_MMS_STATUS_CONTENT_NOT_ACCEPTED:
      TRACE (( 1,"MMS sent status : Content not accepted"));
    break;
    case WIP_MMS_STATUS_UNSUPPORTED_MESSAGE:
      TRACE (( 1,"MMS sent status : Unsupported message"));
    break;
    case WIP_MMS_STATUS_UNSPECIFIED_ERROR:
      TRACE (( 1,"MMS sent status : Unspecified error"));
    break;
    default:
    break;
  }
  wip_mmsClose(p_mmsCtrl);
}
/* Constants */
Const u8 * MSG = "first part";
Const u8 * MSG2 = "second part";
```

```
Const u8 * MSG3 = "last part";

/* Entry point for sending the MMS */
static void appli_entry_point()
{
  u32 date;
  ascii * to_set = "wipsender01@yahoo.fr";
  ascii to_get[50];
  adl_rtcTimeStamp_t RtcTimeStamp;

  /* Get the current time */
  adl_rtcGetTime (&CurTime);

  /* Convert to Time Stamp */
  adl_rtcConvertTime( &CurTime,
                      &RtcTimeStamp,
                      ADL_RTC_CONVERT_TO_TIMESTAMP );
  date = RtcTimeStamp.TimeStamp;
  TRACE (( 1,"[MMS SAMPLE] APPLICATION START"));

  /* Create and initialize the MMS structure*/
  p_mmsCtrl = wip_mmsCreateOpts(
                         WIP_COPT_MMS_DATE,date,
                         WIP_COPT_MMS_STATUS, statuscallback, NULL,
                         WIP_COPT_MMS_SUBJECT,"test",

WIP_COPT_MMS_SENDER_VISIBILITY,WIP_MMS_SENDER_HIDE,

WIP_COPT_MMS_MESSAGE_CLASS,WIP_MMS_MESSAGE_PERSONAL,
                         WIP_COPT_MMS_PRIORITY,WIP_MMS_PRIORITY_NORMAL,
                         WIP_COPT_MMS_FROM,"wiptester01@yahoo.com",
                         WIP_COPT_END);

  /* Add the text part of the MMS */
  wip_mmsAddPart( p_mmsCtrl, MSG, wm_strlen(MSG), WIP_COPT_MMS_PART_TEXT,
                  WIP_COPT_END);
  /* Add the text part of the MMS */
  wip_mmsAddPart( p_mmsCtrl, MSG2, wm_strlen(MSG2), WIP_COPT_MMS_PART_TEXT,
                  WIP_COPT_END);

  /* Set the mail ID of the recipient */
  wip_mmsSetOpts(p_mmsCtrl, WIP_COPT_MMS_TO_MAIL,to_set, WIP_COPT_END);
```

```
  /* Get the already set mail ID of the recipient */
  wip_mmsGetOpts(p_mmsCtrl, WIP_COPT_MMS_TO_MAIL,to_get, WIP_COPT_END);

  /* Add the image part of the MMS */
  wip_mmsAddPart( p_mmsCtrl,
                  image, sizeof(image),
                  WIP_COPT_MMS_PART_JPG, NAME,
                  wm_strlen(NAME), WIP_COPT_END);

  /* Add the audio or sound part of the MMS */
  wip_mmsAddPart( p_mmsCtrl, INTRUDER_ALERT, sizeof(INTRUDER_ALERT),
                  WIP_COPT_MMS_PART_AMR,                         "sound.amr",
wm_strlen("sound.amr"),
                  WIP_COPT_END);

  /* Add the text part of the MMS */
  wip_mmsAddPart( p_mmsCtrl, MSG3, wm_strlen(MSG3), WIP_COPT_MMS_PART_TEXT,
                  WIP_COPT_END);

  /* Remove the text part which is added */
  wip_mmsRemovePart( p_mmsCtrl, MSG2);

  /* Create the HTTP data channel */
  HTTPCnxChannel = wip_HTTPClientCreateOpts( NULL,
                                             NULL,
                                             WIP_COPT_HTTP_PROXY_STRADDR,
                                             MMS_SERVER_IP,
                                             WIP_COPT_HTTP_PROXY_PORT,
                                             MMS_SERVER_PORT,
                                             WIP_COPT_END );
  if( !HTTPCnxChannel ){
    TRACE (( 1,"cannot create HTTP control channel"));
  }
  else{
    /* If the HTTP data channel is created then send the MMS */
    wip_mmsSend( p_mmsCtrl, HTTPCnxChannel, "http://mms1", NULL );
    TRACE (( 1,"MMS sent successfully"));
  }
}
```

15.11. Simple SNMP Example

This example shows how to initialize the SNMP agent.

```
void appli_entry_point( void ) {
int r;
static const u8 engineid[]     = { 0x80,0x00,0x02,0xb8,0x04,0x61,0x62,0x63
};
static const u8 oid_all[]     = { 1 };
static const u8 oidmask_all[] = { 0x80 };

/* Initialising the SNMP agent*/
r = wip_snmpInitOpts(
  WIP_COPT_SYS_NAME, "SNMP TEST",
  WIP_COPT_ENGINE_ID, engineid, sizeof( engineid ),
  /* Two users: admin and guest. Admin requires full authentication and
     Encryption. Guest requires nothing.*/
  /* Option values :user name, authentication, privacy and password */
  WIP_COPT_USER,                 "admin",                 WIP_SNMP_AUTH_MD5,
WIP_SNMP_PRIV_DES,"adminpasswd",
  WIP_COPT_USER, "guest", WIP_SNMP_AUTH_NONE, WIP_SNMP_PRIV_NONE, NULL,
  /* Create a group for each user */
  /* Option values:group name, security model and user name */
  WIP_COPT_GROUP, "admin_group", 3, "admin", /* SNMPv3 security model */
  WIP_COPT_GROUP, "guest_group", 1, "guest", /* SNMPv1 has no security */
  WIP_COPT_GROUP, "public_group", 1, "public",
  WIP_COPT_GROUP, "public_group", 2, "public",
  /* One view representing the whole tree */
  /* Option values :view name, exclude, oid, oid mask and oid length */
  WIP_COPT_VIEW, "all", FALSE, oid_all, oidmask_all, 1,
  /* Access: admin_group users will have access to everything
     read-write-notify, guest_group will have read-only access */
  /* Option values :group name, context, context is prefix, security model,
     security level, read, write, notify */
  WIP_COPT_ACCESS, "admin_group", "", FALSE, 3, WIP_SNMP_SECLVL_AUTHPRIV,
  "all", "all", "all",
  WIP_COPT_ACCESS, "guest_group", "", FALSE, 0, WIP_SNMP_SECLVL_NOAUTH,
  "all", NULL,  NULL,
  WIP_COPT_ACCESS, "public_group","", FALSE, 0, WIP_SNMP_SECLVL_NOAUTH,
  "all", NULL,  NULL,
  WIP_COPT_COMMUNITY, "public", "0.0.0.0", "0.0.0.0", "public",
  WIP_COPT_END );
```

```
    if( OK == r ) {
      TRACE (( 1,"[SAMPLE] SNMP engine successfully launched"));
    }
    else{
      TRACE (( 1,"[SAMPLE] Cannot initialize SNMP"));
    }
}
```

15.12. Simple Finalizer Example

This example shows how to use the finalizer function.

```
static wip_channel_t some_connected_socket = NULL;
/* Finalizer function which is called when the channel is closed */
static void my_finalizer( void *ctx ) {
  TRACE (( 1,"finalization msg: the socket has now been
             completely released"));
}
void entry_point() {
  /* Function that creates the channel with the option to pass the
finalizer
     function to the channel */
  some_tcp_connected_socket = wip_TCPClientCreateOpts(
                              "www.example.com", 80,
                              evh, NULL,
                              WIP_COPT_FINALIZER, my_finalizer,
                              WIP_COPT_END );
}
/* Function to remove the finalizer */
void suppress_finalization_msg() {
  wip_setOpts( some_tcp_connected_socket,
               WIP_COPT_FINALIZER, NULL,
               WIP_COPT_END );
}
```

15.13. Simple IP TUN/TAP Channel Example

This example shows how to use the IP TUN/TAP channel.

```
#define LEN  800
/* IP TUN/TAP channel event handler */
void tunHandler( wip_event_t *ev, void *ctx )
{
  s32 ret;
  u8 pkt[1500];
  switch( ev->kind ) {
    /* These events are always generated at creation of channel */
    case WIP_CEV_OPEN:
    case WIP_CEV_WRITE:
     break;
    case WIP_CEV_READ:
    /* Read the packets that have been received */
    while( ( len = wip_read( ev->channel, pkt, sizeof( pkt ) ) > 0 ) {
      /* Process the received packet */
    }
    break;
    default:
    break;
  }
}
/* IP channel initialization */
void tunnel_init( void )
{
  wip_channel_t tun;
  wip_in_addr_t sym_addr, sym_net, sym_mask;
  /* enable IP forwarding */
  wip_netSetOpts( WIP_NET_OPT_IP_FORWARD, TRUE,
                  WIP_NET_OPT_END );
  wip_inet_aton( "192.168.1.44", &sym_addr );
  wip_inet_aton( "192.168.0.0", &sym_net );
  wip_inet_aton( "255.255.255.0", &sym_mask );
  /* create an IP TUN/TAP channel */
```

```
  tun = wip_TUNCreate ( sym_addr, 0, tunHandler, NULL );
  /* Forward traffic destinated to 192.168.0.0/24 to the IP TUN/TAP */
  wip_ipRouteAdd( sym_net, sym_mask, sym_addr );
…
  u8 buffer[LEN];
  /* Inject IP packet */
  wip_write( tun, buffer, LEN );
}
```

16. Error Codes

16.1. IP Communication Library Initialization and Configuration Error Codes

Error Code	Error Value	Description
WIP_NET_ERR_NO_MEM	-20	Memory allocation error
WIP_NET_ERR_OPTION	-21	Invalid option
WIP_NET_ERR_PARAM	-22	Invalid option value
WIP_NET_ERR_INIT_FAILED	-23	Initialization failed

16.2. Bearer Service Error Codes

Error Code	Error Value	Description
WIP_BERR_NO_DEV	-20	The device does not exist
WIP_BERR_ALREADY	-21	The device is already opened
WIP_BERR_NO_IF	-22	The network interface is not available
WIP_BERR_NO_HDL	-23	No free handle
WIP_BERR_BAD_HDL	-24	Invalid handle
WIP_BERR_OPTION	-25	Invalid option
WIP_BERR_PARAM	-26	Invalid option value
WIP_BERR_OK_INPROGRESS	-27	Connection started, an event will be sent after completion
WIP_BERR_BAD_STATE	-28	The bearer is not stopped
WIP_BERR_DEV	-29	Error from link layer initialization
WIP_BERR_NOT_SUPPORTED	-30	Not a GSM bearer
WIP_BERR_LINE_BUSY	-31	Line busy
WIP_BERR_NO_ANSWER	-32	No answer
WIP_BERR_NO_CARRIER	-33	No carrier
WIP_BERR_NO_SIM	-34	No SIM card inserted
WIP_BERR_PIN_NOT_READY	-35	PIN code not entered
WIP_BERR_GPRS_FAILED	-36	GPRS setup failure
WIP_BERR_PPP_LCP_FAILED	-37	LCP negotiation failure
WIP_BERR_PPP_AUTH_FAILED	-38	PPP authentication failure
WIP_BERR_PPP_IPCP_FAILED	-39	IPCP negotiation failure
WIP_BERR_PPP_LINK_FAILED	-40	PPP peer not responding to echo requests
WIP_BERR_PPP_TERM_REQ	-41	PPP session terminated by peer
WIP_BERR_CALL_REFUSED	-42	Incoming call refused

16.3. Channel Error Codes

Error Code	Error Value	Description
WIP_CERR_ABORTED	-1000	Tried to read/write an aborted TCP client.
WIP_CERR_CSTATE	-999	The channel is not in WIP_CSTATE_READY state.
WIP_CERR_NOT_SUPPORTED	-998	The option is not supported by channel.
WIP_CERR_OUT_OF_RANGE	-997	The option value is out of range.
WIP_CERR_MEMORY	-996	adl_memGet() memory allocation failure.
WIP_CERR_INTERNAL	-995	Internet Library internal error (probable bug, shouldn't happen).
WIP_CERR_INVALID	-994	Invalid option or parameter value.
WIP_CERR_DNS_FAILURE	-993	Couldn't resolve a name to an IP address.
WIP_CERR_RESOURCES	-992	No more TCP buffers available.
WIP_CERR_PORT_IN_USE	-991	TCP server port already used.
WIP_CERR_REFUSED	-990	TCP connection refused by server.
WIP_CERR_HOST_UNREACHABLE	-989	No route to host.
WIP_CERR_NETWORK_UNREACHABLE	-988	No network reachable at all.
WIP_CERR_PIPE_BROKEN	-987	TCP connection broken.
WIP_CERR_TIMEOUT	-986	Timeout (for DNS request, TCP connection, PING response…)

16.4. SMTP Error Codes

When an internal or protocol error occurs during the SMTP session, an event WIP_CEV_ERROR will be received in the event handler. The function wip_getOpts can be used along with WIP_COPT_ERROR option to determine the cause of an error. The error code returned can have either positive or negative value depending on the cause of an error.

- a negative error code indicates that it is a Internet Library socket error code
- a positive error code indicates that it is an error from the SMTP protocol

Below example shows how to retreive error code using wip_getOpts function.

```
u32 ErrorCode = 0;
u32 StatusCode = 0;
ascii **pErrorString;

ret = wip_getOpts(Channel,
                  WIP_COPT_ERROR, &ErrorCode,
                  WIP_COPT_END);
if (ErrorCode > 0)
{
  // SMTP lib: get the protocol status error code + string
  ret = wip_getOpts(Channel,
                    WIP_COPT_SMTP_STATUS_CODE, &StatusCode, &ErrorString,
                    WIP_COPT_END);
}
else
{
  // Internet Library socket error
}
```

16.4.1. Internet Library SMTP Library Error Code

When the returned ErrorCode is positive, it is either a protocol error or a Internet Library SMTP library error.

The Internet Library SMTP library error codes are:

```
typedef enum {
  // No error:
  WIP_SMTP_ERR_NONE = 0,            // No Internet Library SMTP error
  // Internet Library SMTP LIB error:
  // (a protocol status code is available with the WIP_COPT_ERROR)
  WIP_SMTP_ERR_SERV_NO_READY,       // Connect attempt to server failed
  WIP_SMTP_ERR_AUTH_LOGIN,          // AUTH LOGIN command passed  error
  WIP_SMTP_ERR_AUTH_USER,           // Authentication Username error
  WIP_SMTP_ERR_AUTH_PASS,           // Authentication Password error
  WIP_SMTP_ERR_MAIL_FROM,           // Sender error
  WIP_SMTP_ERR_RCPT,                // Recipients error
  WIP_SMTP_ERR_RCPT_CC,             // CC Recipients error
  WIP_SMTP_ERR_RCPT_BCC,            // BCC Recipients error
  WIP_SMTP_ERR_DATA_INPUT,          // Start send DATA error
  WIP_SMTP_ERR_DATA_SENT            // End send DATA error <crlf>.<crlf>
} wip_smtpClientError_e;
```

Common error codes from the SMTP protocol are:

- 535 Authentication failed
- 251 User not local
- 450 Requested mail action not taken: mailbox unavailable
- 553 Requested action not taken: mailbox name not allowed
- 500 Syntax error, command unrecognized
- 501 Syntax error in parameters or arguments
- 502 Command not implemented
- 503 Bad sequence of commands

For other codes, refer to the RFC2821 of the SMTP protocol.

16.4.2. Internet Library Library Error Codes

When the returned code is negative, it is a Internet Library socket error.

```
typedef enum wip_error_t {
  WIP_CERR_OK = 0,
  WIP_CERR_ABORTED = -1000,
  WIP_CERR_CSTATE,
  WIP_CERR_NOT_SUPPORTED,
  WIP_CERR_OUT_OF_RANGE,
  WIP_CERR_MEMORY,
  WIP_CERR_INTERNAL,
  WIP_CERR_INVALID,
  WIP_CERR_DNS_FAILURE,
  WIP_CERR_RESOURCES,
  WIP_CERR_PORT_IN_USE,
  WIP_CERR_REFUSED,
  WIP_CERR_HOST_UNREACHABLE,
  WIP_CERR_NETWORK_UNREACHABLE,
  WIP_CERR_PIPE_BROKEN,
  WIP_CERR_TIMEOUT,
  WIP_CERR_LAST = WIP_CERR_TIMEOUT
} wip_error_t;
```

16.5. POP3 Error Codes

When an internal or protocol error occurs during the POP3 session, an event WIP_CEV_ERROR will be received in the event handler. The function wip_getOpts can be used along with WIP_COPT_ERROR option to determine the cause of an error. The error code returned can have either positive ornegative value depending on the cause of an error.

- a negative error code indicates that it is a Internet Library socket error code
- a positive error code indicates that it is an error from the POP3 protocol

Below example shows how to retreive error code using wip_getOpts function.

```
u32 ErrorCode = 0;
u32 StatusCode = 0;
ascii **pErrorString;
ret = wip_getOpts(Channel,
                  WIP_COPT_ERROR, &ErrorCode,
                  WIP_COPT_END);
if (ErrorCode > 0)
{
  // POP3 lib:/ get protocol status error + string
  ret = wip_getOpts(Channel,
                    WIP_COPT_POP3_STATUS_CODE, &StatusCode, &ErrorString,
                    WIP_COPT_END);
}
else
{
  // Internet Library socket error
}
```

16.5.1. Internet Library POP3 Library Error Code

When the returned ErrorCode is positive, it is either a protocol error or a Internet Library POP3 library error.

The Internet Library POP3 library error codes are:

```
typedef enum {
  // No error:
  WIP_POP3_ERR_NONE = 0,       // No Internet Library POP3 error
  // Internet Library POP3 LIB error:
  // (a protocol status code is available with the WIP_COPT_ERROR)
                               // Connect attempt to server failed
WIP_POP3_ERR_SERV_NO_READY,
  WIP_POP3_ERR_AUTH_LOGIN,     // AUTH LOGIN command passed error
  WIP_POP3_ERR_AUTH_USER,      // Authentication Username error
  WIP_POP3_ERR_AUTH_PASS,      // Authentication Password error
  WIP_POP3_ERR_STAT,           // Get total mails and size error
  WIP_POP3_ERR_LIST,           // Try listing mail inventory error
  WIP_POP3_ERR_RETR,           // Try retrieving mail error
  WIP_POP3_ERR_DELE            // Try deleting mail error
} wip_pop3ClientError_e;
```

Common error codes from the POP3 protocol are:

- +1: Protocol error due to “-ERR” received in protocol
- 0: equivalent to protocol “+OK”

For other codes, refer to the RFC1939 of the POP3 protocol.

16.5.2. Internet Library Library Error Codes

When the returned code is negative, it is a Internet Library socket error.

```
typedef enum wip_error_t {
  WIP_CERR_OK = 0,
  WIP_CERR_ABORTED = -1000,
  WIP_CERR_CSTATE,
  WIP_CERR_NOT_SUPPORTED,
  WIP_CERR_OUT_OF_RANGE,
  WIP_CERR_MEMORY,
  WIP_CERR_INTERNAL,
  WIP_CERR_INVALID,
  WIP_CERR_DNS_FAILURE,
  WIP_CERR_RESOURCES,
  WIP_CERR_PORT_IN_USE,
  WIP_CERR_REFUSED,
  WIP_CERR_HOST_UNREACHABLE,
  WIP_CERR_NETWORK_UNREACHABLE,
  WIP_CERR_PIPE_BROKEN,
  WIP_CERR_TIMEOUT,
  WIP_CERR_LAST = WIP_CERR_TIMEOUT
} wip_error_t;
```

Index

SIERRA
WIRELESS