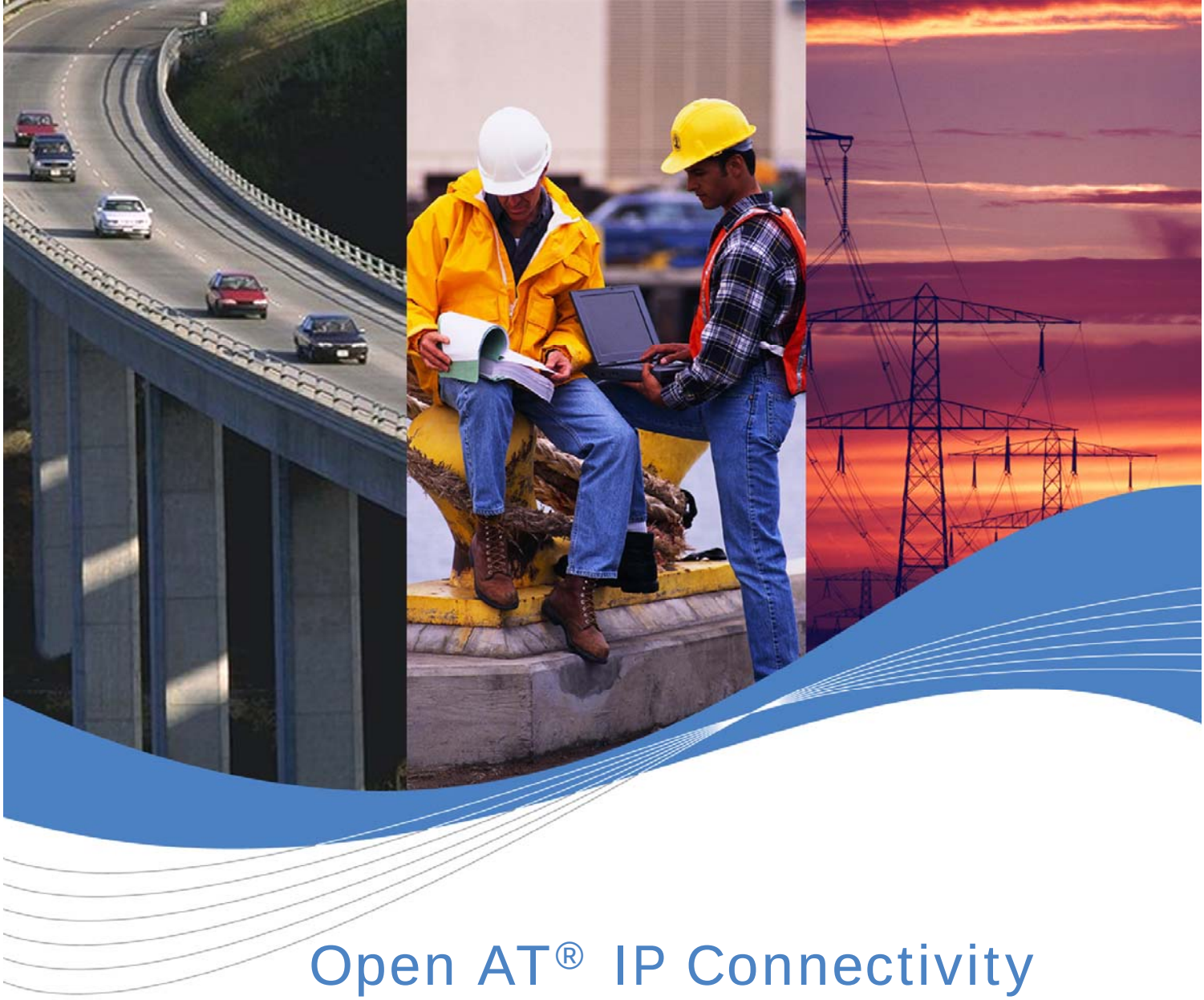

Open AT® IP Connectivity Development Guide (WIPlib 1.10)

Revision: 003
Date: September 2006

OPEN AT

wavecom
Make it wireless

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

Open AT® IP Connectivity Development Guide (WIPlib V1.10)

Reference:	WM_DEV_OAT_UGD_021
Revision:	003
Date:	22 September 2006

Trademarks

 ®, WAVECOM®, WISMO®, Open AT® and certain other trademarks and logos appearing on this document, are filed or registered trademarks of Wavecom S.A. in France or in other countries. All other company and/or product names mentioned may be filed or registered trademarks of their respective owners.

Copyright

Overview

The aim of this document is to provide Wavecom customers with a full description of the APIs associated with the Open AT® IP Connectivity library.

Document History

Level	Date	History of the evolution	Writer
001	22 May 2006	Creation	Wavecom
002	07 August 2006	Preliminary	Wavecom
003	22 September 2006	Final	Wavecom

Contents

List of Figures

1 Introduction

1.1 Related Documents

None.

1.2 Abbreviations and Glossary

1.2.1 Abbreviations

Abbreviation	Description
ADL	Application Development Layer
API	Application Programming Interface
APN	Access Point Name
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
IGMP	Internet Group Management Protocol
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communication
HTTP	Hyper Text Transfer Protocol
ICMP	Internet Control Message Protocol

Abbreviation	Description
IMAP	Internet Message Access Protocol
IN/OUT/GLB	In, Out or Global. See Glossary.
IP	Internet Protocol
IPCP	Internet Protocol Control Protocol
LCP	Link Control Protocol
M	Mandatory
MS-CHAP	Microsoft Challenge Handshake Authentication
MS	Mobile Station
MSS	Maximum Segment Size
NA	Not Applicable
NU	Not Used
O	Optional
PAP	Password Authentication Protocol
PDP	Packet Data Protocol
POP3	Post Office Protocol
POSIX	Portable Operating System Interface
PPP	Point-to-Point Protocol
RFC	Request For Comments

Abbreviation	Description
SMS	Short Messaging Service
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
TOS	Type Of Service
TTL	Time To Live
UART	Universal Asynchronous Receiver Transmitter
UDP	User Data Protocol
USB	Universal Serial Bus
WIFI	Wireless Fidelity
3G	The third generation of developments in wireless technology

1.3 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.

Wavecom 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 WIP 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

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

Figure 1: Communication between Four Equipments

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

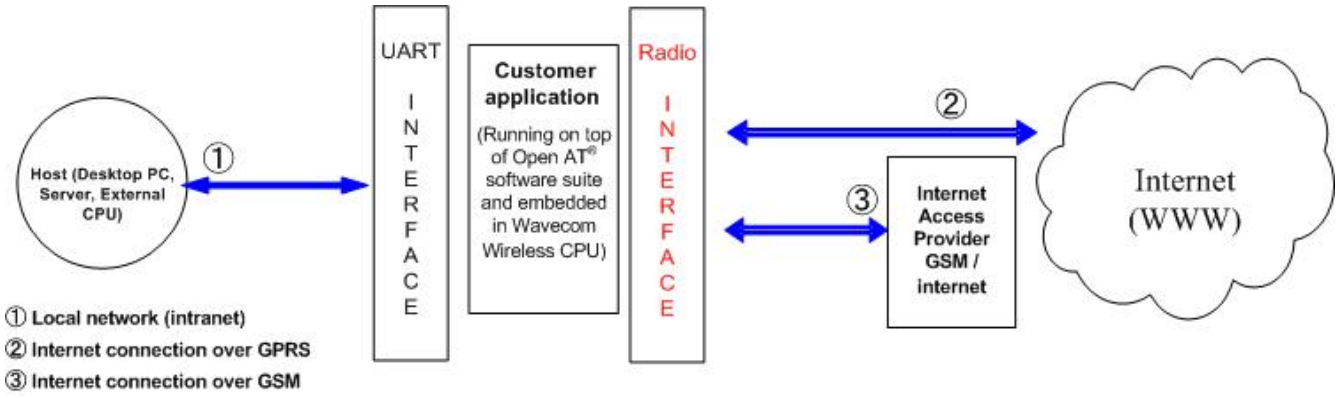

Figure 2: Uses of the New IP Stack (Use Cases ② and ③ are Exclusive)

Open AT® 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

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 Open AT® users who communicate with IP, using GPRS, serial links, or any IP-compatible physical peripherals (WIFI, Ethernet) or radio bearers (EDGE, 3G) supported by Wavecom wireless CPUs.

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.

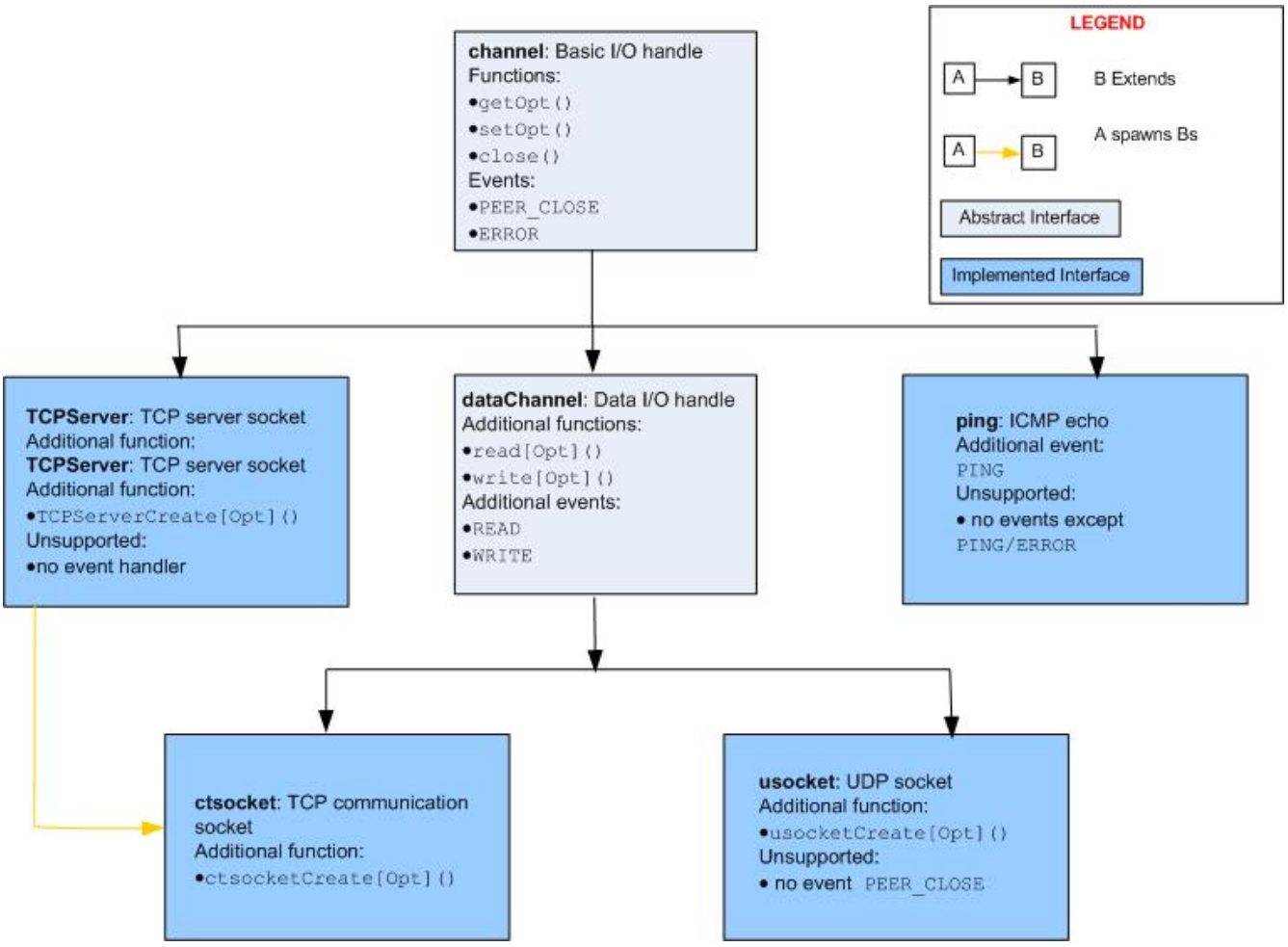

Figure 3: Channel Classes Hierarchy

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 a `u32 readable` field indicating the number of readable bytes, and `WRITE` has a `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

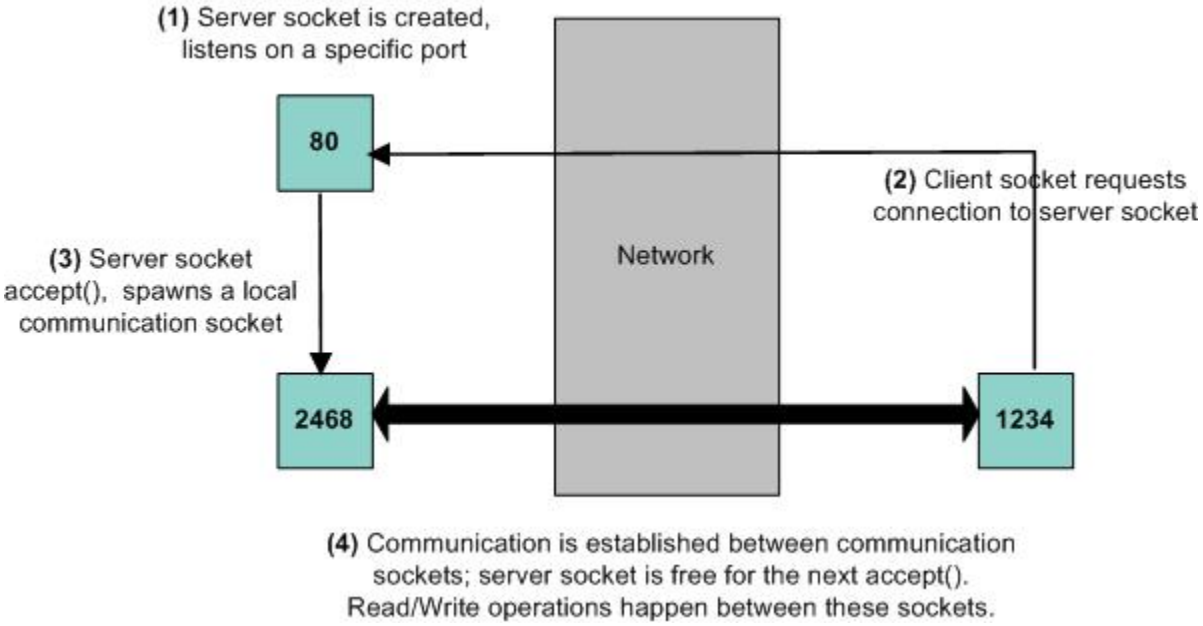

Figure 4: TCP Socket Spawning Process

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 support the `create`, `getopt`, `setopt` and `close` operations.

2.5.3.1 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 `TCPClients` 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 to close 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.

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. The size of the different element is as follows:

Option	Size in bytes
WIP_NET_OPT_SOCK_MAX	380
WIP_NET_OPT_BUF_MAX	1544
WIP_NET_OPT_IP_ROUTE_MAX	24
WIP_NET_OPT_RSLV_QUERY_MAX	128
WIP_NET_OPT_RSLV_CACHE_MAX	224

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() otherwise, the module restarts

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 sizeof 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_SOCK_MAX	u16	Total number of sockets (UDP and TCP).	8
WIP_NET_OPT_BUF_MAX	u16	Total number of network buffers.	32
WIP_NET_OPT_IP_ROUTE_MAX	u16	Size of IP routing table.	0
WIP_NET_OPT_RSLV_QUERY_MAX	u16	Maximum number of DNS resolver queries.	4
WIP_NET_OPT_RSLV_CACHE_MAX	u16	Size of DNS resolver cache.	4
WIP_NET_OPT_END	none	End of option list.	-

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

Note	This function returns a negative error code WIP_NET_ERR_NO_MEM, only if an application is subscribed to adl_errSubscribe() otherwise, the Wireless CPU 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
WIP_NET_OPT_IP_TOS	u8	Default TOS of outgoing datagrams
WIP_NET_OPT_IP_FRAG_TIMEO	u16	Time to live in seconds of incomplete fragments
WIP_NET_OPT_TCP_MAXINITWIN	u16	Number of segments of initial TCP window
WIP_NET_OPT_TCP_MIN_MSS	u16	Default MSS for off-link connections
WIP_NET_OPT_END	none	End of option list

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

For a list of options followed by pointers to options values, see section on The wip_netSetOpts Function. The list must be terminated by WIP_NET_OPT_END.

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

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 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, Ethernet, 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.

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:

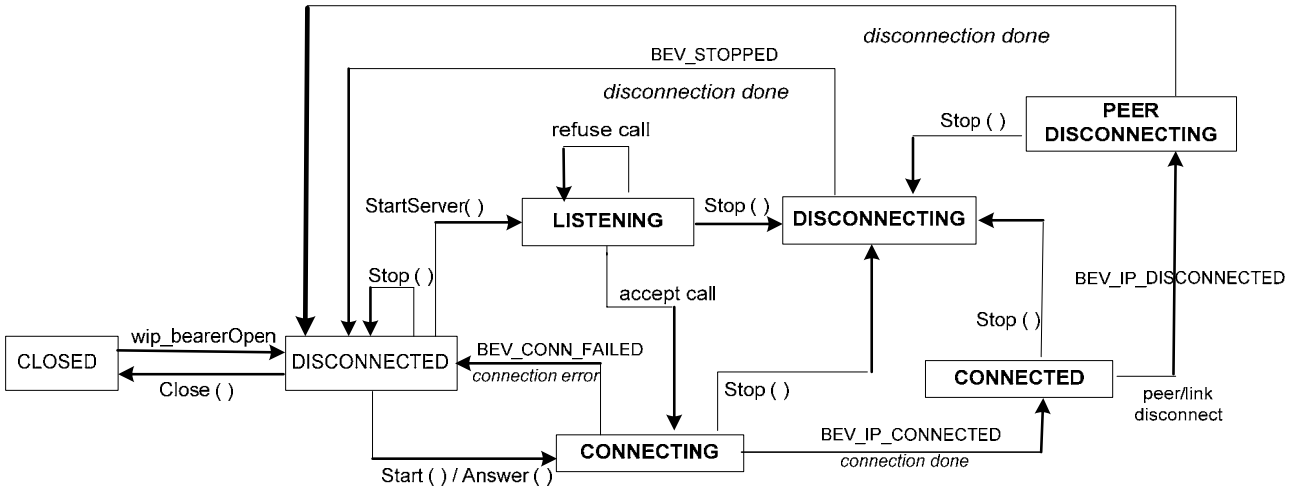

Figure 5: Bearer Management API State Diagram

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

During some transitions, an event is reported to an Open AT® 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

When the bearer is in the Listening state, an Open AT® 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_bearerType_e Type

The **wip_bearerType_e** enumeration stores the type of a bearer.

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

4.3.3 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;
```

4.3.4 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.

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

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

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

 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

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

Option	Value	Description
		interface (get only)
WIP_BOPT_ERROR	s8	Error code indicating the cause of the disconnection (get only)
WIP_BOPT_RESTART	bool	Automatically restart server after connection is terminated
WIP_BOPT_END	none	End of option list
WIP_BOPT_LOGIN	ascii *	Username
WIP_BOPT_PASSWORD	ascii *	Password
Dialing Options		
WIP_BOPT_DIAL_PHONENB	ascii *	Phone number
WIP_BOPT_DIAL_RINGCOUNT	u16	Number of rings to wait before sending the WIP_BEV_DIAL_CALL event
WIP_BOPT_DIAL_MSNULLMODEM	bool	Enable MS-Windows null-modem protocol ("CLIENT"/"SERVER" handshake)
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

Option	Value	Description
WIP_BOPT_PPP_CHAP	bool	Allow CHAP authentication
WIP_BOPT_PPP_MSCHAP1	bool	Allow MSCHAPv1 authentication
WIP_BOPT_PPP_MSCHAP2	bool	Allow MSCHAPv2 authentication
WIP_BOPT_PPP_ECHO	bool	Send LCP echo requests to check if peer is alive
GPRS Options		
WIP_BOPT_GPRS_APN	ascii *	Address of GGSN
WIP_BOPT_GPRS_CID	u8	Cid of the PDP context
WIP_BOPT_GPRS_HEADERCOMP	bool	Enable PDP header compression
WIP_BOPT_GPRS_DATACOMP	bool	Enable PDP data compression
IP Options		
WIP_BOPT_IP_ADDR	wip_in_addr_t	Local IP address
WIP_BOPT_IP_DST_ADDR	wip_in_addr_t	Destination IP address
WIP_BOPT_IP_DNS1	wip_in_addr_t	Address of primary DNS server
WIP_BOPT_IP_DNS2	wip_in_addr_t	Address of secondary DNS server
WIP_BOPT_IP_SETDNS	bool	Configure DNS resolver

Option	Value	Description
		when connection is established
WIP_BOPT_IP_SETGW	bool	Set interface as default gateway when connection is established

4.6.3 Returned Values

The function returns

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

Error Code	Description
WIP_BERR_BAD_HDL	Invalid handle
WIP_BERR_OPTION	Invalid option
WIP_BERR_PARAM	Invalid option value

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 section on The wip_bearerSetOpts Function.

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

Error Code	Description
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

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

Event	Description
	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
- In case of an error, a negative error code as described below:

Error Code	Description
WIP_BERR_OK_INPROGRESS	Connection started, an event will be sent after completion
WIP_BERR_BAD_HDL	Invalid handle
WIP_BERR_BAD_STATE	The bearer is not stopped
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 event list of **wip_bearerStart**

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

Error Code	Description
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);
```

Parameters

br:

In: Bearer handle.

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 is returned to accept the incoming connection, else the call is rejected.

4.10.3 Events

See events of `wip_bearerStart`.

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, the function has no effect.

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

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 bearerInfo_t on success
- a 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 WIP 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.

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 `structs`, 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 paquet in the sent equence */
      u32    response_time; /* Time taken by the echo to come back, in ms. */
      bool   timeout;       /* Did the echo take too long to come bakc? 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

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:

event: The structure describing the event

ctx: An arbitrary pointer to certain user data, generally passed at channel creation time. This allows the user to associate some connection specific contextual data to his channel. If not required. It can obviously be left to NULL.

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

6.1.5 Options

Here is a table which sums up the 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.	-	-
WIP_COPT_KEEPALIVE	Should a dummy packet be sent every two hours to keep the peer TCP socket alive?	bool	bool*
WIP_COPT_SND_BUFSIZE	Size of the emission buffer associated with a socket	u32	u32*
WIP_COPT_RCV_BUFSIZE	Size of the reception buffer associated with a socket	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.	u32	u32*

Option	Description	Set Type	Get Type
WIP_COPT_RCV_LOWAT	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ event.	u32	u32*
WIP_COPT_RCV_TIMEOUT	For PING channels, timeout for ECHO requests.	u32	u32*
WIP_COPT_ERROR	Number of the last error experienced by that socket.	-	s32*
WIP_COPT_NREAD	Number of bytes that can currently be read on that socket.	-	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*

Option	Description	Set Type	Get Type
WIP_COPT_CHECKSUM	Whether the checksum control must be performed by an UDP socket.	bool	bool*
WIP_COPT_NODELAY	When set, TCP packets are sent immediately, even if the buffer is not full enough.	bool	bool*
WIP_COPT_MAXSEG	Maximum size of TCP packets	u32	u32*
WIP_COPT_TOS	Type of Service (cf. RFC 791)	u8	u8*
WIP_COPT_TTL	Time-To-Live for packets sent through this socket; Time-To-Live for this packet, when used in a wip_writeOpts() .	u8	u8*
WIP_COPT_DONTFRAG	If set. UDP datagrams are not allowed to be fragmented when going through the network.	bool	bool*

Option	Description	Set Type	Get Type
WIP_COPT_PEEK	When true, the message is not deleted from the buffer after reading, so that it can be read again.	bool	-
WIP_COPT_PORT	Port occupied by this socket.	u16	u16*
WIP_COPT_STRADDR	Local address of the socket.	ascii*	ascii *buf, u32 buf_len
WIP_COPT_ADDR	Local address of the socket, as a 32 bits integer.	wip_in_addr_t	wip_in_addr_t*
WIP_COPT_PEER_PORT	Port of the peer socket.	u16	u16*
WIP_COPT_PEER_STRADDR	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection.	ascii*	ascii *buf, u32 buf_len
WIP_COPT_PEER_ADDR	Address of the peer socket, as a 32 bits integer.	wip_in_addr_t	wip_in_addr_t *

Option	Description	Set Type	Get Type
WIP_COPT_TRUNCATE	Whether an UDP read operation truncated the received data, due to a lack of buffer space.	bool	bool*
WIP_COPT_REPEAT	Number of PING echo requests to send.	s32	s32*
WIP_COPT_INTERVAL	Time between two PING echo requests, in ms.	u32	u32*
WIP_COPT_SUPPORT_READ	Fails if the channel does not support wip_read() operations. If supported, does nothing.	-	-
WIP_COPT_SUPPORT_WRITE	Fails if the channel does not support wip_write() operations. If supported, does nothing.	-	-

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.

6.2.1.1 Prototype

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

6.2.1.2 Parameters

c:

In: The channel that must be closed.

6.2.1.3 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

6.2.2 The wip_read Function

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

6.2.2.1 Prototype

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

6.2.2.2 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.

6.2.2.3 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.

6.2.3 The wip_readOpts Function

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

6.2.3.1 Prototype

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

6.2.3.2 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.

6.2.3.3 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 section on Options.

6.2.4.1 Prototype

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

6.2.4.2 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.

6.2.4.3 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.

6.2.5 The wip_writeOpts Function

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

6.2.5.1 Prototype

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

6.2.5.2 Parameters

c:

In: The channel to write to.

buffer:

Out: Pointer to the buffer where data to be writen 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.

6.2.5.3 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 section on Options.

6.2.6.1 Prototype

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

6.2.6.2 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.

6.2.6.3 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.

6.2.7 The wip_setOpts Function

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

6.2.7.1 Prototype

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

6.2.7.2 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.

6.2.7.3 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.

6.2.8.1 Prototype

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

6.2.8.2 Parameters

c:

The channel for which the event context must be changed.

ctx:

The new context.

6.2.8.3 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`.

6.2.9.1 Prototype

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

6.2.9.2 Parameter

c:

The channel for which the state must be determined.

6.2.9.3 Returned Values

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

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

6.3.1 Statecharts

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

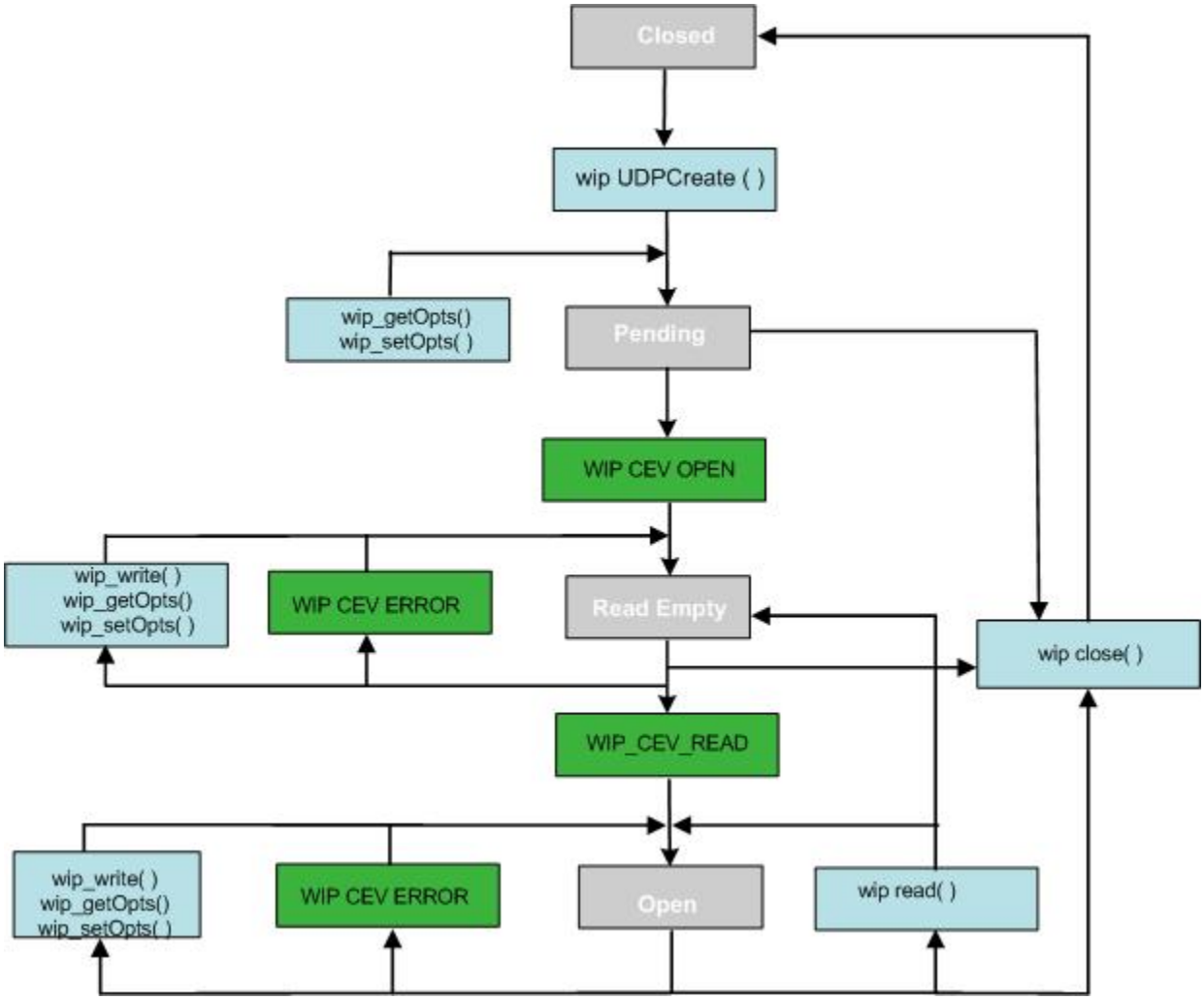

Figure 6: UDP Channel State Diagram

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.

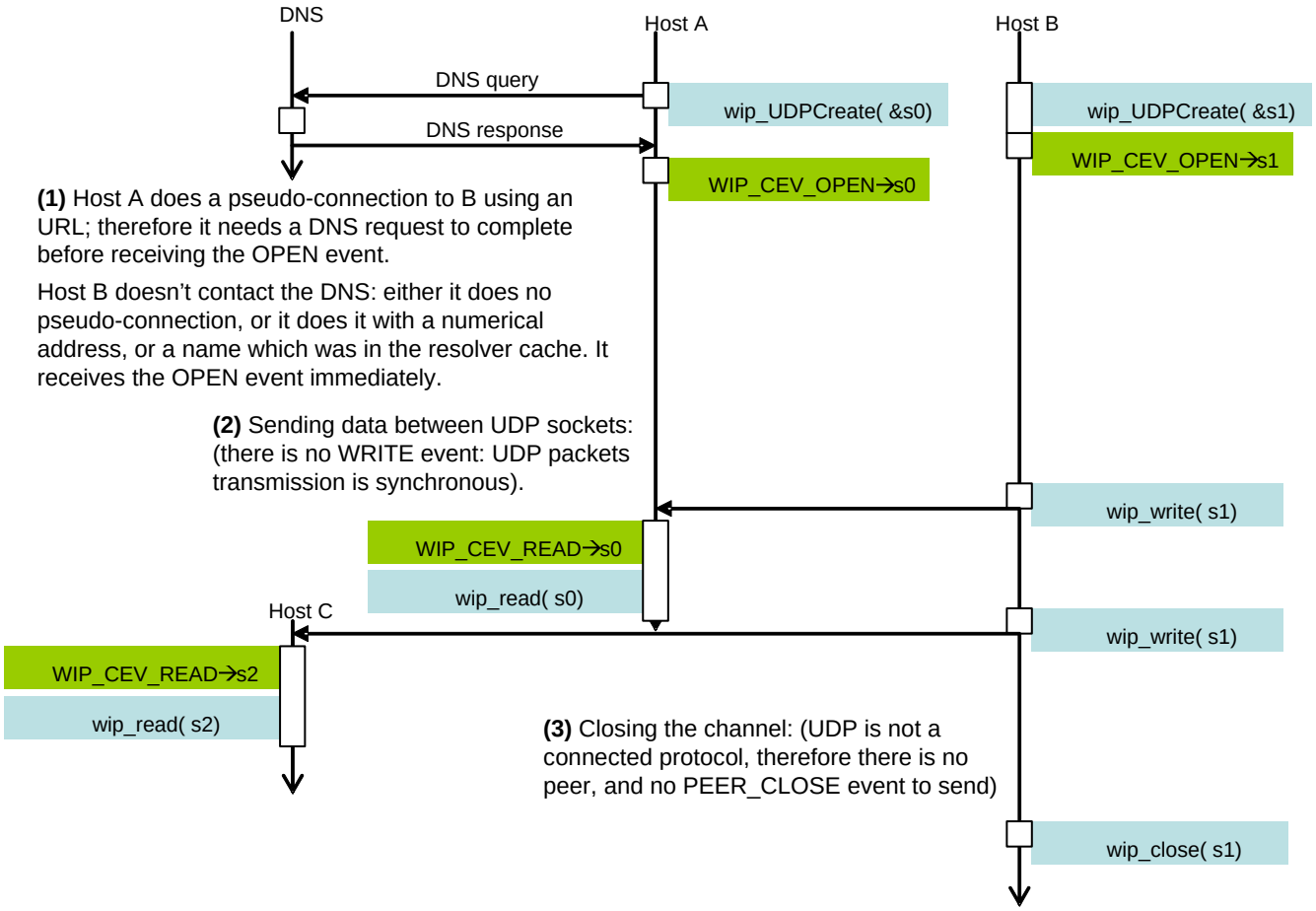

Figure 7: UDP Channel Temporal Diagram

6.3.2 The wip_UDPCreate Function

The **wip_UDPCreate** function creates a channel encapsulating an UDP socket.

6.3.2.1 Prototype

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

6.3.2.2 Parameters

handler:

The event handler which will react to network events happening to this socket. Possible events kinds are WIP_CEV_READ, WIP_CEV_WRITE, WIP_CEV_PEER_CLOSE and WIP_CEV_ERROR. If set to NULL, every event happening to this socket will be discarded.

ctx:

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

6.3.2.3 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.

6.3.3.1 Prototype

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

6.3.3.2 Parameters

handler:

The event handler which will react to network events happening to this socket. Possible event kinds are WIP_CEV_READ, WIP_CEV_WRITE, WIP_CEV_PEER_CLOSE and WIP_CEV_ERROR. If set to NULL, every event happening to 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.
WIP_COPT_RCV_BUFSIZE	<u32>	Size of the reception buffer associated with a socket.

Option	Value	Description
WIP_COPT_CHECKSUM	<bool>	Whether the checksum control must be performed by an UDP socket.
WIP_COPT_TOS	<u8>	Type of Service (cf. RFC 791)
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().
WIP_COPT_DONTFRAG	<bool>	If set. UDP datagrams are not allowed to be fragmented when going through the network.
WIP_COPT_PORT	<u16>	Port occupied by this socket.
WIP_COPT_STRADDR	<ascii*>	Local address of the socket.
WIP_COPT_ADDR	<wip_in_addr_t>	Local address of the socket.
WIP_COPT_PEER_PORT	<u16>	Port of the peer socket.
WIP_COPT_PEER_STRADDR	<ascii*>	Address of the peer socket. If set to NULL on a pseudo-

Option	Value	Description
		connected UDP socket, remove the connection.
WIP_COPT_PEER_ADDR	<wip_in_addr_t>	Address of the peer socket.

6.3.3.3 Returned Values

This function returns:

- the created channel
- NULL on error

6.3.4 The wip_getOpts Options Function for UDP Channels

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

Option	Value	Description
WIP_COPT_END		End of the option
WIP_COPT_SND_BUFSIZE	<u32*>	Size of the emission buffer associated with a socket
WIP_COPT_RCV_BUFSIZE	<u32*>	Size of the reception buffer associated with a socket
WIP_COPT_ERROR	<s32*>	Number of the last error experienced by that socket.
WIP_COPT_NREAD	<u32*>	Number of bytes that can currently be read on that socket.
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.
WIP_COPT_TOS	<u8*>	Type of Service (cf.

Option	Value	Description
		RFC 791)
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().
WIP_COPT_DONTFRAG	<bool*>	If set. UDP datagrams are not allowed to be fragmented when going through the network.
WIP_COPT_PORT	<u16*>	Port occupied by this socket.
WIP_COPT_STRADDR	<ascii* buffer>, <u32 buf_len>	Local address of the socket.
WIP_COPT_ADDR	<wip_in_addr_t*>	Local address of the socket, as a 32 bits integer.
WIP_COPT_PEER_PORT	<u16*>	Port of the peer socket.
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.
WIP_COPT_PEER_ADDR	<wip_in_addr_t*>	Address of the peer socket, as a 32 bits

Option	Value	Description
		integer.
WIP_COPT_SUPPORT_READ		Fails if the channel does not support wip_read() operations. If supported, does nothing.
WIP_COPT_SUPPORT_WRITE		Fails if the channel does not support wip_write() operations. If supported, does nothing.

6.3.5 The wip_setOpts for UDP Channels Function

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

Option	Value	Description
WIP_COPT_END		End of the option
WIP_COPT_SND_BUFSIZE	<u32>	Size of the emission buffer associated with a socket.
WIP_COPT_RCV_BUFSIZE	<u32>	Size of the reception buffer associated with a socket.
WIP_COPT_CHECKSUM	<bool>	Whether the checksum control must be performed by an UDP socket.
WIP_COPT_TOS	<u8>	Type of Service (cf. RFC 791)
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().
WIP_COPT_DONTFRAG	<bool>	If set. UDP datagrams are not allowed to be fragmented when going through the network.

Option	Value	Description
WIP_COPT_PEER_PORT	<u16>	Port of the peer socket.
WIP_COPT_PEER_STRADDR	<ascii*>	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection.
WIP_COPT_PEER_ADDR	<wip_in_addr_t>	Address of the peer socket, as a 32 bits integer.

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.

6.3.6 The wip_readOpts for UDP Channels Function

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

Option	Value	Description
WIP_COPT_END		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.
WIP_COPT_PEER_PORT	<u16*> (get)	Port of the peer socket.
WIP_COPT_PEER_STRADDR	<ascii *buffer>, <u32 buf_len> (get)	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection.
WIP_COPT_PEER_ADDR	<wip_in_addr_t*> (get)	Address of the peer socket, as a 32 bits integer.

6.3.7 The wip_writeOpts for UDP Channels Function

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

Option	Value	Description
WIP_COPT_END		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.
WIP_COPT_PEER_PORT	<u16*> (get)	Port of the peer socket.
WIP_COPT_PEER_STRADDR	<ascii *buffer>, <u32 buf_len> (get)	Address of the peer socket. If set to NULL on a pseudo-connected UDP socket, remove the connection.
WIP_COPT_PEER_ADDR	<wip_in_addr_t*> (get)	Address of the peer socket, as a 32 bits integer.

6.4 TCPServer: Server TCP Sockets

Server TCP 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.

The state diagram is as follows:

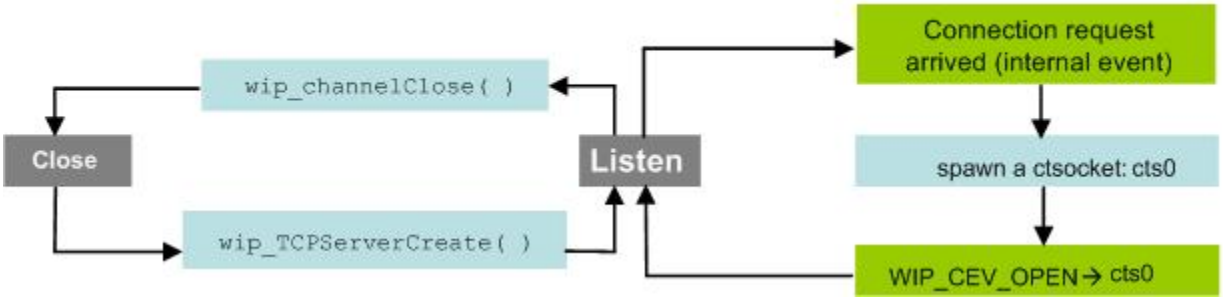

Figure 8: TCP Server Channel State Diagram

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.

6.4.1.1 Prototype

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

6.4.1.2 Parameters

port:

The port number on which TCP server socket listens.

spawnedHandler:

The event handler to be attached to `TCPClients` spawned by this server socket. 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.

6.4.1.3 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.

6.4.2.1 Prototype

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

6.4.2.2 Parameters

port:

The port number on which TCP server socket listens.

spawnedHandler:

The event handler to be attached to TCPClients spawned by this server socket. 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		End of the option
WIP_COPT_SND_BUFSIZE	<u32> (inherited by spawned TCPClients)	Size of the emission buffer

Option	Value	Description
		associated with a socket.
WIP_COPT_RCV_BUFSIZE	<u32> (inherited by spawned TCPClients)	Size of the reception buffer associated with a socket.
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.
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.
WIP_COPT_NODELAY	<bool> (inherited by spawned TCPClients)	When set, TCP packets are sent immediately, even if the buffer is not full enough.
WIP_COPT_TOS	<u8> (inherited by spawned TCPClients)	Type of Service (cf. RFC 791)
WIP_COPT_TTL	<u8> (inherited by spawned TCPClients)	Time-To-Live for packets sent through this socket; Time-To-

Option	Value	Description
		Live for this packet, when used in a wip_writeOpts().

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.

6.4.2.3 Returned Values

This function returns:

- the created channel
- `NULL` on error

6.4.3 The wip_getOpts Options Function for TCPServer Channels

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

Option	Value	Description
WIP_COPT_END		End of the option
WIP_COPT_KEEPALIVE	<bool*>	Should a dummy packet be sent every two hours to keep the peer TCP socket alive?
WIP_COPT_SND_BUFSIZE	<u32*>	Size of the emission buffer associated with a socket.
WIP_COPT_RCV_BUFSIZE	<u32*>	Size of the reception buffer associated with a socket.
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.
WIP_COPT_RCV_LOWAT	<u32*>	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ

Option	Value	Description
		event.
WIP_COPT_NODELAY	<bool*>	When set, TCP packets are sent immediately, even if the buffer is not full enough.
WIP_COPT_TOS	<u8*>	Type of Service (cf. RFC 791)
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().
WIP_COPT_PORT	<u16*>	Port occupied by this socket.
WIP_COPT_STRADDR	<ascii* buff>, <u32 buf_len>	Local address of the socket.
WIP_COPT_ADDR	<wip_in_addr_t*>	Local address of the socket, as a 32 bits integer.

6.4.4 The wip_setOpts Options Function for TCPServer Channels

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

Option	Value	Description
WIP_COPT_END		End of the option
WIP_COPT_KEEPALIVE	<bool>	Should a dummy packet be sent every two hours to keep the peer TCP socket alive?
WIP_COPT_SND_BUFSIZE	<u32> (inherited by spawned TCPClients)	Size of the emission buffer associated with a socket.
WIP_COPT_RCV_BUFSIZE	<u32> (inherited by spawned TCPClients)	Size of the reception buffer associated with a socket.
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.
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

Option	Value	Description
		event.
WIP_COPT_NODELAY	<bool> (inherited by spawned TCPClients)	When set, TCP packets are sent immediately, even if the buffer is not full enough.
WIP_COPT_TOS	<u8> (inherited by spawned TCPClients)	Type of Service (cf. RFC 791)
WIP_COPT_TTL	<u8> (inherited by spawned TCPClients)	Time-To-Live for packets sent through this socket; Time-To-Live for this packet, when used in a wip_writeOpts().

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.

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

The read and write events are received when:

- the data arrives for the first time on the socket (READ event)
- a read attempt returns 0 bytes, or less data than the provided buffer could store (READ event)
- a write attempt writes less than the buffer it had been provided (WRITE event).

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

6.5.2 Statecharts

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

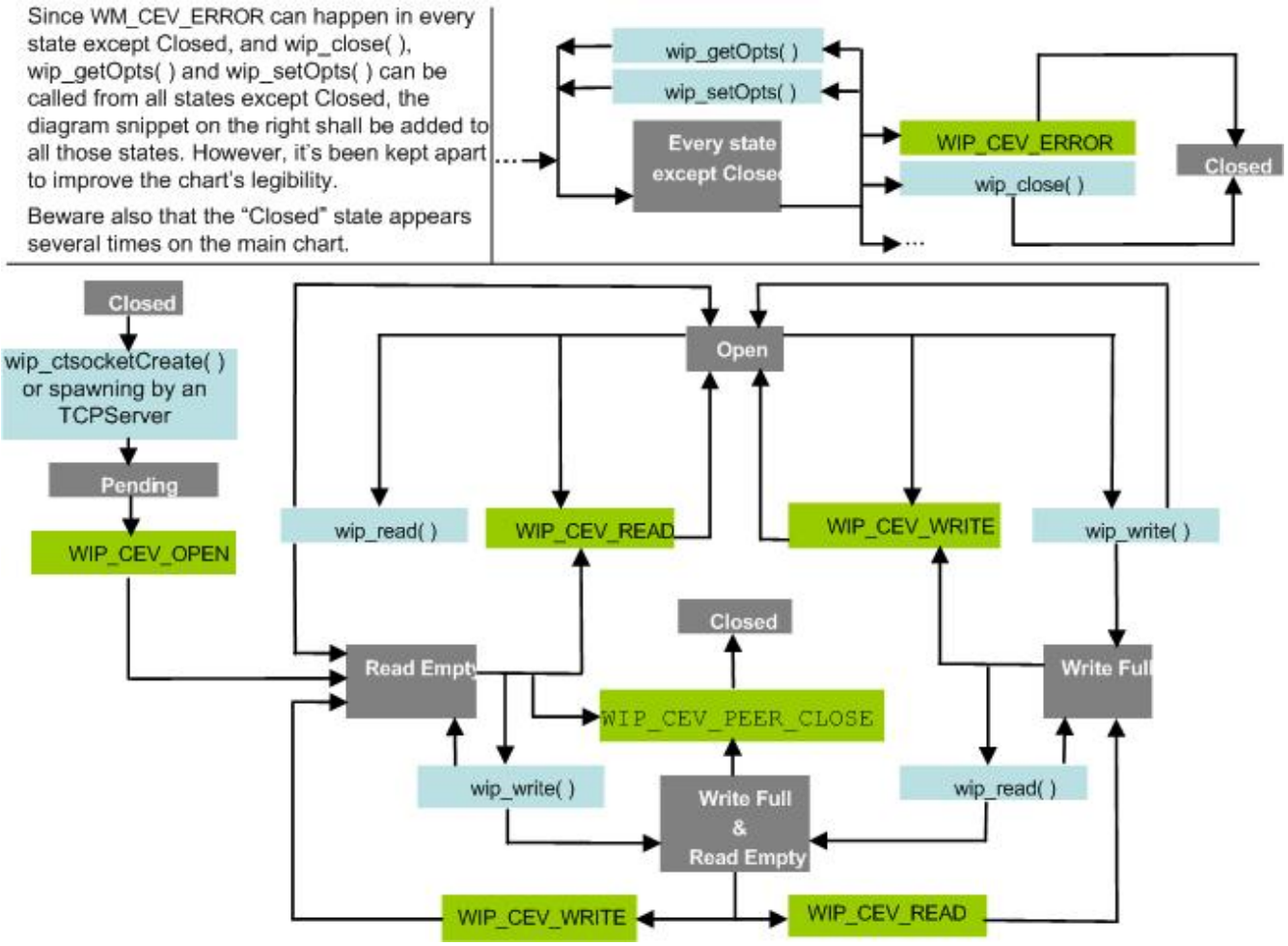

Figure 9: TCP Communication Channel State Diagram

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.

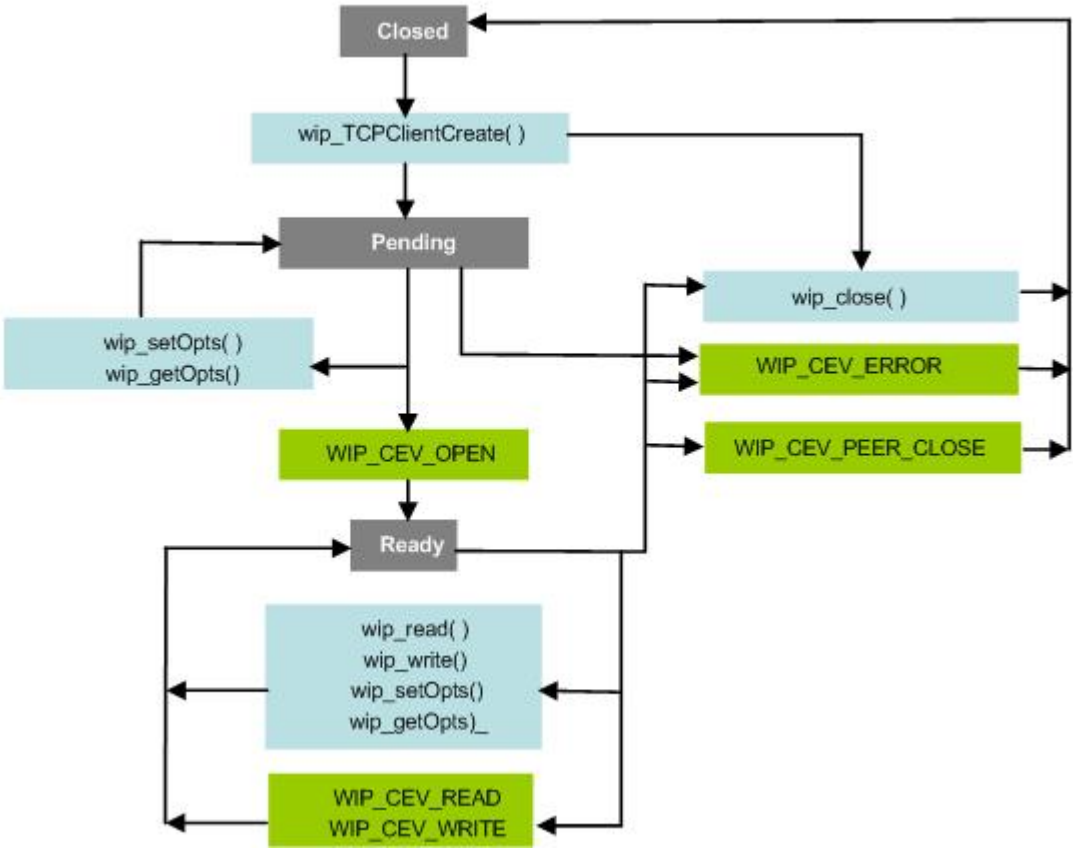

Figure 10: TCP Communication Channel Simplified State Diagram

A typical temporal flow example follows:

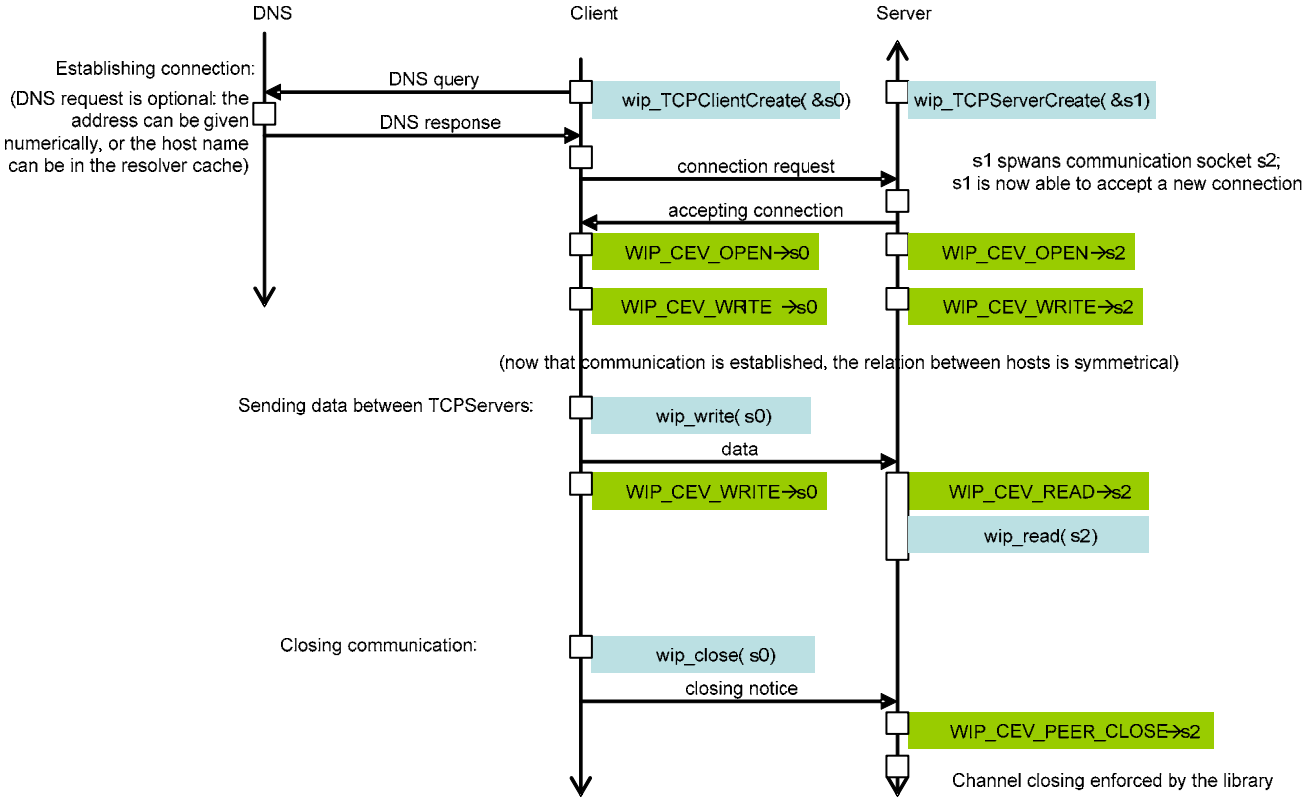

Figure 11: TCP Communication Channel Temporal Diagram

6.5.3 The wip_TCPClientCreate Function

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

6.5.3.1 Prototype

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

6.5.3.2 Parameters

serverAddr:

The address of the destination server, as an ASCII string. 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 event handler which will react to network events happening to this socket. Possible events kinds are WIP_CEV_READ, WIP_CEV_WRITE, WIP_CEV_PEER_CLOSE and WIP_CEV_ERROR. If set to NULL, every event happening to this socket will be discarded.

ctx:

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

6.5.3.3 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.

6.5.4.1 Prototype

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

6.5.4.2 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		End of the option
WIP_COPT_KEEPALIVE	<bool>	Should a dummy packet be sent every two hours to keep the peer TCP socket alive?
WIP_COPT_SND_BUFSIZE	<u32>	Size of the emission buffer associated with a socket.
WIP_COPT_RCV_BUFSIZE	<u32>	Size of the reception buffer associated with a

Option	Value	Description
		socket.
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.
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.
WIP_COPT_NODELAY	<bool>	When set, TCP packets are sent immediately, even if the buffer is not full enough.
WIP_COPT_MAXSEG	<u32>	Maximum size of TCP packets
WIP_COPT_TOS	<u8>	Type of Service (cf. RFC 791)
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().

Option	Value	Description
WIP_COPT_STRADDR	<ascii*>	Local address of the socket.
WIP_COPT_PORT	<u16>	Port occupied by this socket.

6.5.4.3 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.

6.5.5.1 Prototype

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

6.5.5.2 Parameters

c:

The socket that must be aborted.

6.5.5.3 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.

6.5.6.1 Prototype

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

6.5.6.2 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.

6.5.6.3 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 Options Function for TCPClient Channels

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

Option	Value	Description
`WIP_COPT_END`		End of the option
`WIP_COPT_KEEPALIVE`	`<bool*>`	Should a dummy packet be sent every two hours to keep the peer TCP socket alive?
`WIP_COPT_SND_BUFSIZE`	`<u32*>`	Size of the emission buffer associated with a socket.
`WIP_COPT_RCV_BUFSIZE`	`<u32*>`	Size of the reception buffer associated with a socket.
`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.
`WIP_COPT_RCV_LOWAT`	`<u32*>`	Minimum amount of available space that must be available in the reception buffer before triggering a

Option	Value	Description
		WIP_CEV_READ event.
WIP_COPT_ERROR	<s32*>	Number of the last error experienced by that socket.
WIP_COPT_NREAD	<u32*>	Number of bytes that can currently be read on that socket.
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, TCP packets are sent immediately, even if the buffer is not full enough.
WIP_COPT_MAXSEG	<u32*>	Maximum size of TCP packets
WIP_COPT_TOS	<u8*>	Type of Service (cf. RFC 791)
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().

Option	Value	Description
WIP_COPT_PORT	<u16*>	Port occupied by this socket.
WIP_COPT_STRADDR	<ascii* buff>, <u32 buf_len>	Local address of the socket.
WIP_COPT_ADDR	<wip_in_addr_t*>	Local address of the socket, as a 32 bits integer.
WIP_COPT_PEER_PORT	<u16*>	Port of the peer socket.
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.
WIP_COPT_PEER_ADDR	<wip_in_addr_t*>	Address of the peer socket, as a 32 bits integer.
WIP_COPT_SUPPORT_READ		Fails if the channel does not support wip_read() operations. If supported, does nothing.
WIP_COPT_SUPPORT_WRITE		Fails if the channel does not support wip_write() operations. If supported, does nothing.

6.5.8 The wip_setOpts Options Function for TCPClient Channels

The options supported by the **`wip_setOpts`** function, applied to a `TCPClient` are:

Option	Value	Description
`WIP_COPT_END`		End of the option
`WIP_COPT_KEEPALIVE`	`<bool>`	Should a dummy packet be sent every two hours to keep the peer TCP socket alive?
`WIP_COPT_SND_BUFSIZE`	`<u32>`	Size of the emission buffer associated with a socket.
`WIP_COPT_RCV_BUFSIZE`	`<u32>`	Size of the reception buffer associated with a socket.
`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.
`WIP_COPT_RCV_LOWAT`	`<u32>`	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ

Option	Value	Description
		event.
WIP_COPT_NODELAY	<bool>	When set, TCP packets are sent immediately, even if the buffer is not full enough.
WIP_COPT_TOS	<u8>	Type of Service (cf. RFC 791)
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().

6.5.9 The wip_readOpts Options Function for TCPClient Channels

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

Option	Value	Description
`WIP_COPT_END`		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.

6.5.10 The wip_writeOpts Options Function for TCPClient Channels

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

Option	Value	Description
`WIP_COPT_END`		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.

6.6.1 The wip_pingCreate Function

The **wip_pingCreate** function creates a channel supporting a ping session.

6.6.1.1 Prototype

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

6.6.1.2 Parameters

peerAddr:

Address of host user want to ping. This can be either a DNS address, or a numeric one in the form "xxx.xxx.xxx.xxx".

evHandler:

The event handler which will react to network events happening to this socket. Possible events kinds are WIP_CEV_PING and WIP_CEV_ERROR.

ctx:

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

6.6.1.3 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)

6.6.2.1 Prototype

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

6.6.2.2 Parameters

destAddr:

Address of host user want to ping. This can be either a DNS address, or a numeric one in the form "xxx.xxx.xxx.xxx".

handler:

The event handler which will react to network events happening to this socket. Possible events kinds are WIP_CEV_PING and WIP_CEV_ERROR.

ctx:

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		End of the option
WIP_COPT_REPEAT	<s32>	Number of PING echo requests to send.
WIP_COPT_INTERVAL	<u32>	Time between two PING echo requests, in ms.
WIP_COPT_RCV_TIMEOUT	<u32>	For PING channels, timeout for ECHO requests.
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().
WIP_COPT_NWRITE	<u32>	Number of bytes that can currently be written on that socket. For a PING, size of the request (default=20)

6.6.2.3 Returned Values

This function returns:

- the created channel on success
- NULL on error

6.6.3 The wip_getOpts Options Function for ping Channels

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

Options	Value	Description
WIP_COPT_END		End of the option
WIP_COPT_REPEAT	<s32*>	Number of PING echo requests to send.
WIP_COPT_INTERVAL	<u32*>	Time between two PING echo requests, in ms.
WIP_COPT_RCV_TIMEOUT	<u32*>	For PING channels, timeout for ECHO requests.
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().
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 Options Function for ping Channels

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

Options	Value	Description
`WIP_COPT_END`		End of the option
`WIP_COPT_INTERVAL`	<u32>	Time between two PING echo requests, in ms.
`WIP_COPT_RCV_TIMEOUT`	<u32>	For PING channels, timeout for ECHO requests.
`WIP_COPT_REPEAT`	<s32>	Number of PING echo requests to send.
`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().
`WIP_COPT_NWRITE`	<u32>	Number of bytes that can currently be written on that socket. For a PING, size of the request (default=20)

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`, 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

It will also support `wip_write()`, so that the application can actually send the data which represent the file contents; 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 support `wip_read()` and `wip_close()`.

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

It might seem surprising that both connection channels and data transfer channels are supported by the same `wip_channel_t C` type; this is due to the lack of subtyping ability of C. 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:

The connection channel

file_name:

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:

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:

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

7.2.3 Returned Values

The function returns data transfer channel.

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.

7.3.3 Returned Values

The function returns data transfer channel.

.

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 a file content. Instead, this function create and return dedicated data transfer channel, which support 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:

The connection channel

file_name:

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:

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

ctx:

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

7.4.3 Returned Values

The function returns data transfer channel.

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 a file content. Instead, this function create and return dedicated data transfer channel, which support 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.

7.5.3 Returned Values

The function returns data transfer channel.

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 thre connection channel on which file will be deleted.

name:

Name of the file to delete.

7.8.3 Returned Values

The function returns:

- 0 on success
- negative error code on 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:

Connection channel on which file will be deleted

name:

Name of the directory to delete

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:

Connection channel on which file will be renamed old_name

old_name:

Previous name of the file.

new_name:

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 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.

7.11.1 Prototype

```
int wip_getFileSize(
                wip_channel_t cx,
                ascii *name);
```

7.11.2 Parameters

ftp_cx:

Connection channel the file whose size is required

name:

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 WIP 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 {
  ascii name [WIP_FILE_NAME_MAX];
  u32 size;
  int is_dir: 1; /*true if the name corresponds to a directory*/
  int can_read : 1; /*true if the file can be read*/
  int can_write: 1; /*true if the file can be written*/
} wip_fileInfo_t;
```

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. When all fileInfo structures are read, an event `WIP_CEV_PEER_CLOSE` will be received in the event handler and `wip_read()` returns zero.

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 signalled by a new `WIP_CEV_READ` event
- a `WIP_CEV_PEER_CLOSE` after the last data is arrived

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:

Connection channel

dir_name:

Name of the directory whose content must be listed (can be NULL, in this case the CWD wil be listed)

evh:

Event handler which will receive the events

ctx:

evh user data.

7.12.3 Returned Values

The function returns spawned transfer channel.

8 FTP Client

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

- it is based on Wavecom'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 data transfer sessions can happen simultaneously, which means that the application can read/write several files concurrently.

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: 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	char	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

Option	Value	Description
		Default is "wipftp@wavecom.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	Plug-in handling the results from the LIST FTP command (non-standard, server-dependent)

8.3.3 Returned Values

The function returns:

- the created channel on success
- NULL on error

8.4 The wip_setOpts Function for FTP Client

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		End of the option
WIP_COPT_KEEPALIVE	<bool>	Should a dummy packet be sent every two hours to keep the peer TCP socket alive?
WIP_COPT_SND_BUFSIZE	<u32>	Size of the emission buffer associated with a socket.
WIP_COPT_RCV_BUFSIZE	<u32>	Size of the reception buffer associated with a socket.
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.
WIIP_COPT_RCV_LOWAT	<u32>	Minimum amount of available space that must be available in the reception buffer before triggering a WIP_CEV_READ event.
WIP_COPT_NODELAY	<bool>	When set, TCP packets are sent immediately, even if the buffer is not full enough.
WIP_COPT_TOS	<u8>	Type of Service (cf. RFC 791)
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().
WIP_COPT_FTP_TYPE	<char>	TBD
WIP_COPT_PASSIVE	<bool>	Active or Passive Default is passive mode

WIP_COPT_LIST_PLUGIN	<wip_eventHandler_f>	Plug-in handling the results from the LIST FTP command (non-standard, server-dependent)

8.5 The wip_getOpts Function for FTP Client

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		End of the option
WIP_COPT_KEEPALIVE	<bool*>	Should a dummy packet be sent every two hours to keep the peer TCP socket alive?
WIP_COPT_SND_BUFSIZE	<u32*>	Size of the emission buffer associated with a socket.
WIP_COPT_RCV_BUFSIZE	<u32*>	Size of the reception buffer associated with a socket.
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.
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.
WIP_COPT_ERROR	<s32*>	Number of the last error experienced by that socket.
WIP_COPT_NREAD	<u32*>	Number of bytes that can currently be read on that socket.

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, TCP packets are sent immediately, even if the buffer is not full enough.
WIP_COPT_TOS	<u8*>	Type of Service (cf. RFC 791)
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().
WIP_COPT_PORT	<u16*>	Port occupied by this socket.
WIP_COPT_STRADDR	<ascii* buff>, <u32 buf_len>	Local address of the socket.
WIP_COPT_ADDR	<wip_in_addr_t*>	Local address of the socket, as a 32 bits integer.
WIP_COPT_PEER_PORT	<u16*>	Port of the peer socket.
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.
WIP_COPT_PEER_ADDR	<wip_in_addr_t*>	Address of the peer socket, as a 32 bits integer.
WIP_COPT_SUPPORT_READ		Fails if the channel does not support wip_read() operations. If supported, does nothing.

WIP_COPT_SUPPORT_WRITE		Fails if the channel does not support wip_write() operations. If supported, does nothing.
WIP_COPT_FTP_TYPE	<char>	TBD
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>	Plug-in handling the results from the LIST FTP command (non-standard, server-dependent)

8.6 The wip_close Function for FTP Client

The FTP session is closed with **`wip_close`** function. Refer section 6.2.1 for more details on this function.

8.7 The wip_getFile Function for FTP Client

The function **`wip_getFile`** is used to download a file from the FTP server. Refer section 7.2 for more details on this function.

8.8 The wip_getFileOpts Function for FTP Client

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_OFFSET	u32 n	Restart the transfer at the nth byte
WIP_COPT_END		End of the option

Refer section 7.3 for more details on wip_getFile function.

8.9 The wip_putFile Function for FTP Client

The **wip_putFile** function is used to upload a file to the FTP server. Refer section 7.4 for more details on **wip_putFile** function.

8.10 The wip_putFileOpts Function for FTP Client

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_OFFSET	u32 n	Restart the transfer at the nth byte
WIP_COPT_END		End of the option

Refer section 7.5 for more details on wip_getFile function.

9 HTTP Client

HTTP client provides an application interface for generating HTTP requests using Wavecom TCP/IP implementation (WIP plug-in). It is based on WIP 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 channel with `wip_HTTPCreate()` or `wip_HTTPCreate()` that will store information common to all further HTTP requests: http version, address of proxy server, some HTTP request headers, this channel will also maintain persistent connections. A new channel is then created for each HTTP request using `wip_getFile()` or `wip_putFile()`.

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,
   WIP_HTTP_VERSION_1_1
} wip_httpVersion_e;
```

The WIP_HTTP_VERSION_1_0 constant indicates HTTP 1.0.

The WIP_HTTP_VERSION_1_1 constant indicates HTTP 1.1.

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,
   WIP_HTTP_METHOD_HEAD,
   WIP_HTTP_METHOD_POST,
   WIP_HTTP_METHOD_PUT,
   WIP_HTTP_METHOD_DELETE,
   WIP_HTTP_METHOD_TRACE,
   WIP_HTTP_METHOD_CONNECT
} wip_httpMethod_e;
```

9.4 The wip_httpHeader_t type

The wip_httpHeader_t structure defines a HTTP header field.

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

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 event handler which will react to events happening to this channel. Currently no event is defined so it can be set to NULL.

ctx:

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

9.5.3 Returned Values

- the created 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_HTTPClientCreate (
                                    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 option names. The option names must be followed by option values. 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.
WIP_COPT_SND_BUFSIZE	<u32>	This option sets the size of the TCP socket send buffer.
WIP_COPT_PROXY_STRADDR	<ascii *>	This option sets the hostname of the HTTP proxy server, a NULL value disables the proxy server.
WIP_COPT_PROXY_PORT	<u16>	This option sets the

Option	Value	Description
		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.
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), a NULL value can be passed to remove a previously defined header field.
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.

9.6.3 Returned Values

- the created channel on success
- NULL on error.

9.7 The wip_getFile Function for HTTP Client

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 thanks to the WIP_COPT_HTTP_METHOD option.

When HTTP 1.1 is used, a new TCP channel is not created for each request destinated to the same server or proxy, instead the TCP connection is maintained by the HTTP session whenever possible.

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

Event	Description
WIP_CEV_OPEN	This event is sent when the the response message header has been received. The wip_getOpts function can be used to retrieve 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.
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 occured.

Refer section 7.3 for more details on wip_getFile function.7.2

9.8 The wip_getFileOpts Function for HTTP Client

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 section 9.7

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.

The wip_getFileOpts Function for HTTP Client

Option	Value	Description
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.

9.9 The wip_putFile Function for HTTP Client

The **`wip_putFile`** function sends a HTTP PUT request to the given URL. For more details on this function, refer section 7.4

Note	The only difference with `wip_getFile` is the default HTTP method

9.10 The wip_putFileOpts Function for HTTP Client

The **`wip_putFileOpts`** function sends a HTTP PUT request to the given URL with the user defined options. For more details on this function, refer section 7.5

9.11 The wip_read Function for HTTP Client

The **wip_read** function is used to read the response message body. This function is not supported by session channels.

For more details on this function, refer section 6.2.2.

9.12 The wip_write Function for HTTP Client

The **`wip_write`** function is used to write the request message body. Not all requests have a message body. This function is not supported by session channels.

For more details on this function, refer section 6.2.4

9.13 The wip_shutdown Function for HTTP Client

The **wip_shutdown** function is used on a request channel to signals the end of the message body, it has no effect if the request has no message body. This function can also be used to skip data of the response message.

This function is not supported by session channels.

For more details on this function, refer section 6.5.6

9.14 The wip_setOpts Function for HTTP Client

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, see wip_HTTPClientCreate for a description of supported options.

For more details on this function, refer section 6.2.7

9.15 The wip_getOpts Function for HTTP Client

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

Session channels support 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.
WIP_COPT_SND_BUFSIZE	<u32 *>	This option returns the current size of the TCP socket send buffer.
WIP_COPT_PROXY_STRADDR	<ascii *>,<u32>	This option returns the hostname of the HTTP proxy server.
WIP_COPT_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 channels support 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.

9.16 The wip_abort Function for HTTP Client

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

For more details on this function, refer section 6.5.5

9.17 The wip_close Function for HTTP Client

On the session channel the **`wip_close`** function aborts any current request and release resources associated with the session channel.

Note	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 reseted to indicates to the server that the request was incomplete.

For more details on this function, refer section 6.2.1

10 Examples of Application

10.1 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...
0     */
    break;
  case WIP_BEV_IP_DISCONNECTED:
    /*
     * stop IP application...
     */
    break;
  /* other events: */
  default:
    /*
     * cannot start bearer: report error to higher levels...
     */
    break;
```

```
  }
}
/* bearer handle */
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) {
    /* cannot start bearer */
    bearerClose( myBearer);
    return FALSE;
  }

  /* connection status will be reported to the event handler */
  return TRUE;
}
```

10.2 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.

10.2.1 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     "Wavecom"
#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);
}
```

10.2.2 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:
      printf( "Received greeting from server: %s\n", buffer);
      wip_write( c, "NAME", strlen( "NAME"));
      *ctx = FORENAME_REQUEST_SENT;
      break;
    case FORENAME_REQUEST_SENT:
      printf( "Forename:\t%s\n", buffer);
      wip_write( c, "FORENAME", strlen( "FORENAME"));
      *ctx = NAME_REQUEST_SENT;
      break;
    case NAME_REQUEST_SENT:
      printf( "Name:\t%s\n", buffer);
      wip_write( c, "PHONE", strlen( "PHONE"));
      *ctx = PHONE_REQUEST_SENT;
      break;
    case PHONE_REQUEST_SENT:
      printf( "Phone:\t%s\n", buffer);
      wip_write( c, "QUIT", strlen( "QUIT"));
      *ctx = QUIT_REQUEST_SENT;
      break;
    case QUIT_REQUEST_SENT:
      printf( "Server says goodbye:\t%s\n", buffer);
      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);
}
```

10.3 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`).

```
#define CX_NUM      16   /* How many connection can be handled simultaneously */
#define SERVER_PORT 1235 /* Port number of the server */


/* 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 {
  s32 cx_id;            /* Number identifying the connection */
  enum {
    FREE,               /* This context is currently unused */
    WAIT_FOR_SECOND_CX, /* One connection has been made, waiting for the second */
    CONNECTED           /* Both clients are connected, they can chat together */
  } state;
  wip_channel_t cx0;       /* First client to connect */
  wip_channel_t cx1;       /* Second client to connect */
} 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;
      if( ev->content.read.readable < sizeof( id)) return; // wait for more data
      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 = malloc( readable);
    wip_read( c, buffer, readable);
    wip_write( dual, buffer, readable);
    free( 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);
}
```

10.4 Simple FTP Example

This program downloads a file named data.bin from the server ftp.wavecom.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.wavecom.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 WIP 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.

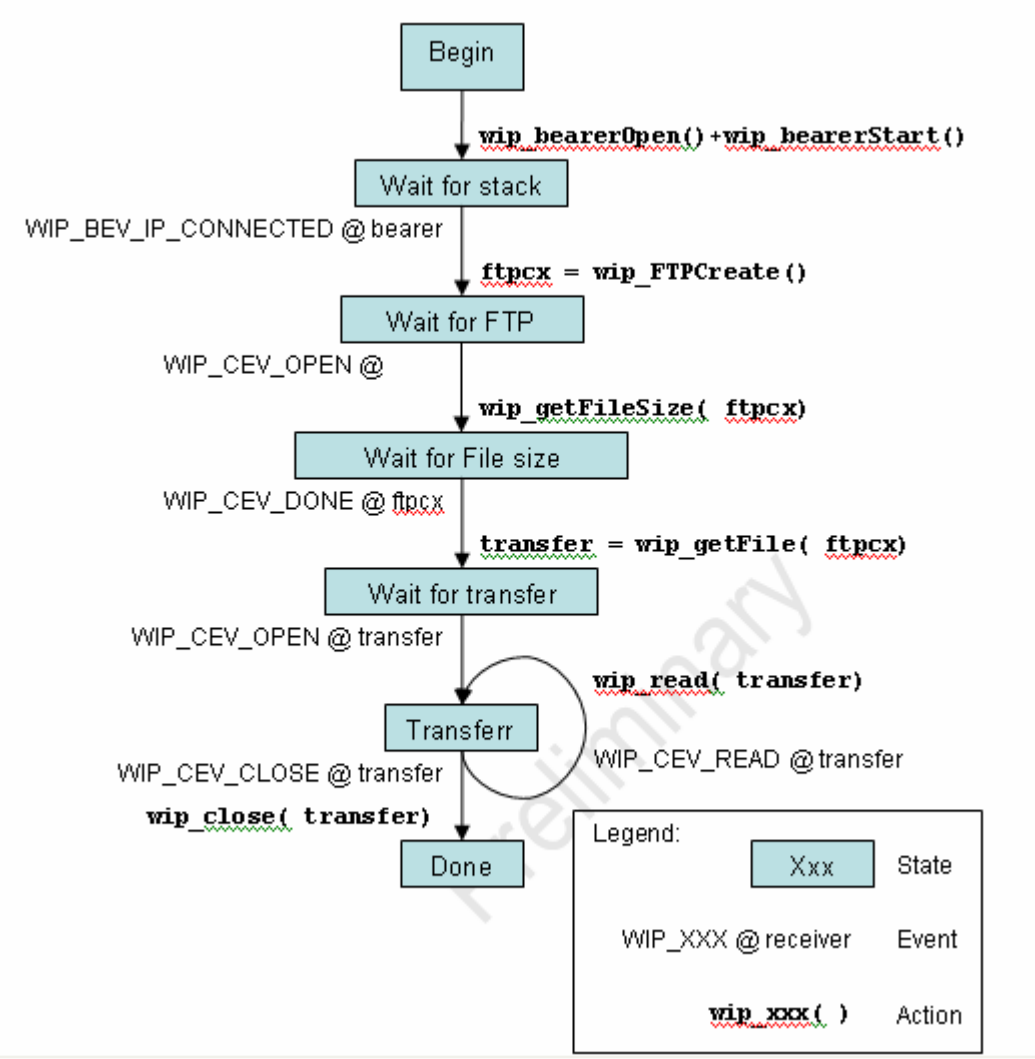

Figure 12: State machine of a simple FTP application

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();
        state = STATE_YYY3; /* relevant state transition. */

        break;
      case STATE_YYY1:
        someOtherAction();
        state = STATE_YYY2;
        break;
      /* etc. */
    }

 case WIP_CEV_XXX1: switch( state) {
       /* etc. */
```

```
    }
    /* etc. */
  }
}
```

10.5 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

10.6 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;

  /* get originating channel */ ch = ev->channel;

  switch( ev->kind) {
  case WIP_CEV_OPEN:
    /* get status code */
    wip_getOpts( ch,
                 WIP_COPT_HTTP_STATUS_CODE, &ret,
                 WIP_COPT_END);
    if( ret != 200) {
      /* not OK... */
    }
    break;

  case WIP_CEV_READ:
    /* read html page */
```

```
    while( (ret = wip_read( ch, buf, sizeof( buf))) > 0) {
      /* ...process html data... */
    }
    break;

  case WIP_CEV_PEER_CLOSE:
    /* html page has been received */
    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.wavecom.com",
                   http_event, NULL,
                   WIP_COPT_HTTP_HEADER, "Accept", "text/html",
                   WIP_COPT_END);
}
```

11 Error Codes

11.1 IP Communication Plug-In 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

11.2 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

Bearer service error codes

Error Code	Error Value	Description
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

11.3 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	WIP 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.

Error Code	Error Value	Description
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...)

wavecom

Make it wireless

WAVECOM S.A. - 3 esplanade du Foncet - 92442 Issy-les-Moulineaux Cedex - France - Tel: +33(0)1 46 29 08 00 - Fax: +33(0)1 46 29 08 08
Wavecom, Inc. - 4810 Eastgate Mall - Second Floor - San Diego, CA 92121 - USA - Tel: +1 858 362 0101 - Fax: +1 858 558 5485
WAVECOM Asia Pacific Ltd. - 4/F, Shui On Centre - 6/8 Harbour Road - Hong Kong - Tel: +852 2824 0254 - Fax: +852 2824 025