institut de recherche en informatique
et systèmes aléatoires

WULL6: A WINDOWS UDP-LITE LIBRARY FOR IPV6

USER'S GUIDE

Version: 1.0.0.7

Date: 2007-02-22

Auteur(s):

Laurent Guillo

Adrien Schadle

Table of contents

1. WHAT IS WULL6? 5

2. UDP-LITE A QUICK OVERVIEW 6

3. A FIRST EXAMPLE 8

4. OVERALL ARCHITECTURE 10

5. ADDRESSING 12

6. WULL6 API 13

6.1. WULL66BIND 13

6.2. WULL66CLEANUP 14

6.3. WULL66CLOSEEVENT 14

6.4. WULL66CLOSESOCKET 15

6.5. WULL66CONNECT 15

6.6. WULL6CREATEEVENT 16

6.7. WULL6EVENTSELECT 17

6.8. WULL6GETLASTERROR 17

6.9. WULL6GETSOCKNAME 18

6.10. WULL6GETSOCKOPT 19

6.11. WULL6IOCTLSOCKET 20

6.12. WULL6RECV 21

6.13. WULL6RECVFROM 23

6.14. WULL6RESETEVENT 25

6.15. WULL6SEND 25

6.16. WULL6SENDTO 27

6.17. WULL6SETSOCKOPT 29

6.18. WULL6SOCKET 30

6.19. WULL6STARTUP 31

7. INSTALLATION AND CONFIGURATION 33

7.1. INSTALLATION 33

7.2. CONFIGURATION 33

7.3. RESTRICTION OF USE 33

ANNEXE A A FULL RECEIVER EXAMPLE 35

ANNEXE B ERROR CODES 41

1. What is Wull6?

Wull6[1] is Windows DLL[2], which implements the UDP-Lite[3] protocol for IPv6 in unicast mode only. Wull6 provides developers with specific UDP-Lite sockets they can use in the same way they use classic UDP sockets.

UDP-Lite is similar to UDP. However, UDP-Lite allows applications to specify the checksum coverage. Hence, developers can take advantage of Wull6 API to specify that value.

The present user guide includes five main chapters:

- **UDP-Lite: a quick overview** which details differences between UDP and UDP-Lite and how the checksum coverage is implemented.
- **A first example** which explains how to use the Wull6 API.
- **General architecture** outlines how Wull6 implements UDP-Lite.
- **Wull6 API** describes all functions Wull6 provides developers with.
- **Installation and configuration** lists all the required steps to install and to set Wull6.

These chapters are followed by annexes that contain a full sender/receiver example with an error bit generator, a few figures about performances and error codes.

---

[1] Wull6 : UDP-Lite library for IPv6.

[2] Dynamic-linked library.

[3] UDP-Lite is described in RFC 3828.

2. UDP-Lite a quick overview[4]

There is class of applications that benefit from having damaged data delivered rather than discarded by the network. A number of codecs for voice and video fall into this class. These codecs may be designed to cope better with errors in the payload than with loss of entire packets.

The generally available transport protocol best suited for these applications is UDP. However, in IPv4, the UDP checksum covers either the entire packet or nothing at all. In IPv6, the UDP checksum is mandatory and must not be disabled since the IP header has no checksum anymore.

UDP-Lite is a new transport protocol that has been designed to meet these error tolerant applications' needs. Indeed, UDP-Lite provides a checksum with an optional partial coverage. When using this option, a packet is divided into a sensitive part (covered by the checksum) and an insensitive part (not covered by the checksum). Only errors in the sensitive part cause the packet to be discarded by the transport layer.

When the checksum covers the entire packet[5], UDP-Lite is semantically identical to UDP.

A UDP header and a UDP-Lite header have the same size and all their fields are the same except one.

A UDP packet has the following structure:

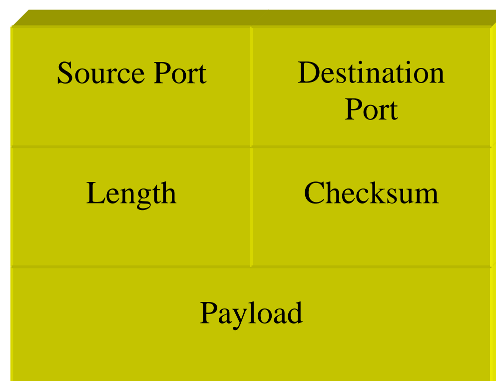

Figure 1: UDP packet format

The field "Length" equals the total length: header and payload. The checksum is computed from UDP data and UDP header but also from fields belonging to the IP packet. These fields are gathered in the IP pseudo header:

- 128 bit IP source address field,
- 128 bit IP source destination field,

---

[4] Most of this UDP-Lite description is extracted from the RFC 3828.

[5] which is Wull6's default behaviour.

- 8 bit zero field,
- 8 bit Protocol field,
- 16 bit UDP length field.

A UDP-Lite packet has the following format:

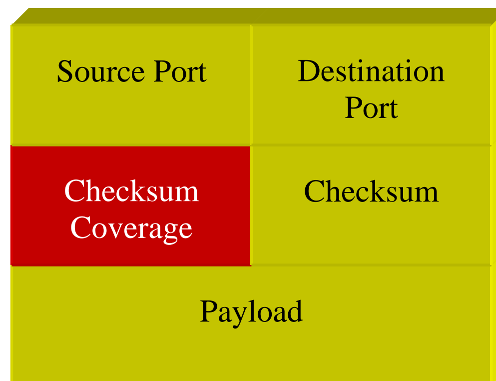

Figure 2 : UDP-Lite packet format

Its format differs from UDP in that the length field has been replaced with a checksum coverage field.

Checksum Coverage is the number of bytes, counting from the first byte of the UDP-Lite header, that are covered by the checksum. The UDP-Lite header must always be covered by the checksum. A checksum coverage of zero indicates that the entire UDP-Lite packet is covered by the checksum. The checksum coverage is 0 or at least 8. A receiver discards UDP-Lite packets with a Checksum coverage value of 1 to 7 and also packets with a checksum coverage value greater than the IP length.

3. A first example

The following source code is a snippet of a whole program that can send UDP-Lite packets.

It is a program that sends a file with UDP-Lite and is waiting for reception of another one. The Wull API calls are in bold type.

```
// initialize wull6
if ((nRet = WULL6Startup() != 0))
{
	printf("ERROR: WULL6Startup failed with error %d\n", nRet);
	return 0;
}

// create a wull6 socket
sMySocket=WULL6Socket(AF_INET6,SOCK_DGRAM, IPPROTO_UDPLITE);


// get the structure for the destination address
struct addrinfo aiHints;
struct addrinfo *aiList = NULL;		// list od addrinfo supported by the host
memset(&aiHints, 0, sizeof(aiHints));
aiHints.ai_family = AF_INET6;
aiHints.ai_flags = AI_NUMERICHOST | AI_PASSIVE;

if ((nRet = getaddrinfo(REMOTE_IPV6_ADDRESS ,REMOTE_PORT, &aiHints, &aiList)) != 0)
{
	// error
	return -1;
}
else
{
	// specify a checksum coverage of 8
	short optchecksum = 8;
	nRet = WULL6setsockopt(sMySocket, IPPROTO_UDPLITE,
UDPLITE_CHECKSUM_COVERAGE,
				(char*)&optchecksum,  sizeof(optchecksum));

	// send a UDP-Lite packet
	nRet = WULL6SendTo(sMySocket, cPayload, sizeof(cPayload), 0, aiList->ai_addr,aiList-
>ai_addrlen);

}

// close the socket
WULL6CloseSocket(sMySocket);

// free the wull6 library
```

This example outlines how to:

- Initialize the Wull6 API,
- Create a UDP-Lite socket,
- Set the checksum coverage,

- Send data,
- Stop the Wull6 Interface.

The required steps are described hereafter. First, a Windows developer has already noticed that functions belonging to the Wull6 API start with the string "WULL66". Moreover, they are similar to Windows socket API functions.

The first step is to initialize the Wull6 API with the "WULL66Startup" function. Without this call, all following calls to the other Wull6 functions will fail[6].

Wull6 allows developers to create sockets with UDP-Lite as protocol. To do so, a new value for the "WULL66Socket"'s protocol argument is added: IPPROTO_UDPLITE.

To specify the checksum coverage, Wull takes advantage of the "WULL66setsockopt" which has a specific optname, UDPLITE_CHECKSUM_COVERAGE, for the level IPPROTO_UDPLITE. Hence, the checksum coverage can be specified as an argument of the "WULL66Setsockopt" function.

Sending a packet is similar to the Windows "classic" socket way. The function "WULL66SendTo" used in the example or the other sending function "WULL66Send" are analogous to Windows' ones.

As the Wull6 API has been initialized, it must be properly stopped. That is why the application calls "WULL66Cleanup".

All functions of the Wull6 API can fail. To get more information about why, call the "WULL66GetLastError" function[7].

[6] WULL6Startup may return an error, cf. 6.1.

[7] Error codes are listed in Annexe B.

4. Overall architecture

Wull6 implements UDP-Lite as a Windows DLL written in C++. It is shipped with three main files:

- **Wull6.h**, a header file you must include in your application source code,
- **Wull6.lib**, a file you must link with,
- **Wull6.dll**, the Wull DLL.

The following diagram describes the overall architecture:

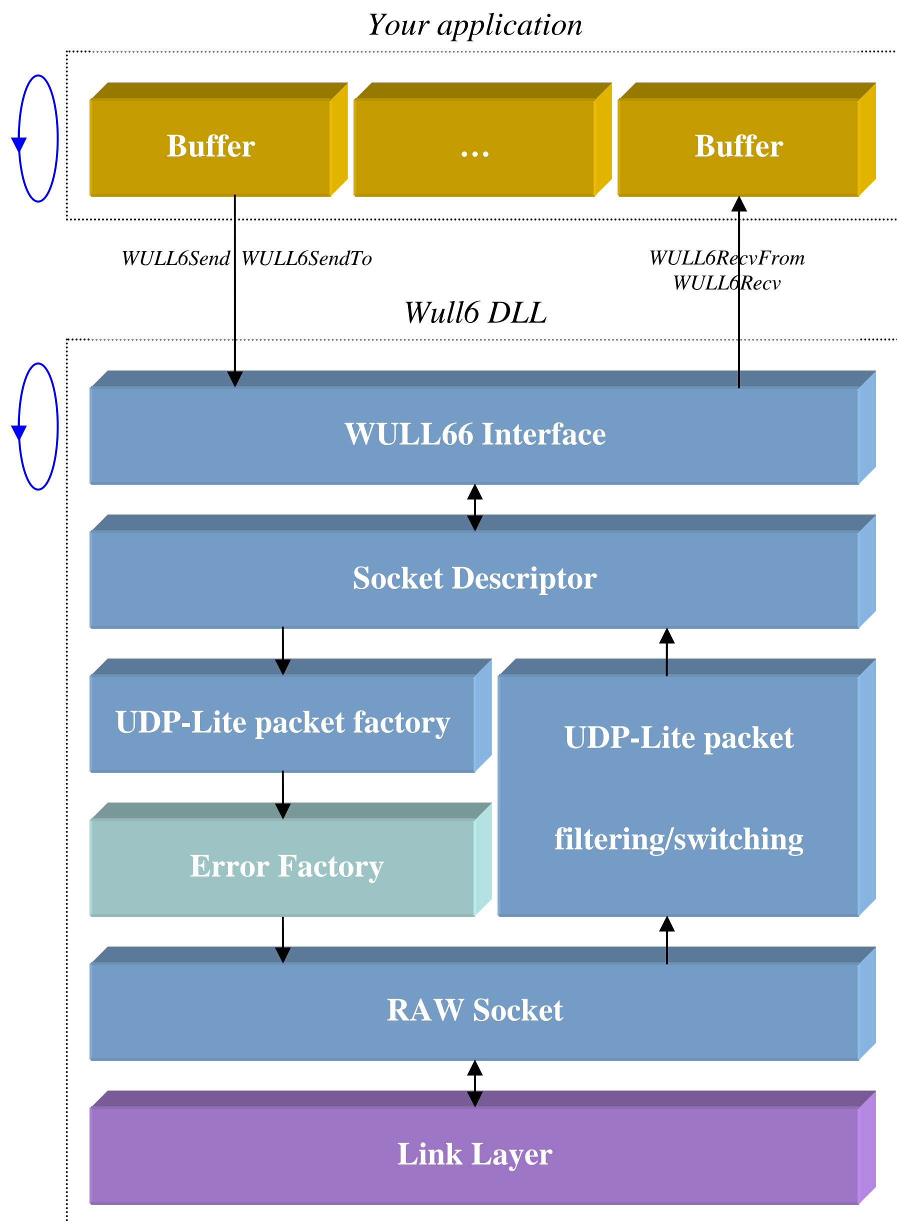

Figure 3: Wull6's overall architecture

This diagram depicts two threads, “your application” and “wull6.dll”, and the link layer. The application, which could be multi-threaded also, calls Wull6 functions via the Wull6 interface. Most of its calls are related to sending and receiving actions.

When Wull6 receives a sending request, data are included in a whole IPv6/UDP-Lite packet: IPv6 and UDP-Lite headers are added and the UDP-Lite checksum is computed. This is the UDP-Lite packet factory’s role. Then, a thread handling a raw socket is told it can send this packet.

When the raw socket receives an incoming packet, several controls are made (e.g. is an UDP-Lite packet for me, with correct checksum, …) and the incoming packet passed the tests, data are copied in the related socket’s buffer. The control module is the UDP-Lite packet filtering/switching module. Then, the application can read its data (an event can be set if the application was not in blocking mode).

The module “Error Factory” is optional. It introduces bit errors once IP/UDP-Lite packet are completely built and ready to send. It must only be activated for test purposes[8].

As the DLL makes use of socket in raw mode, it only runs on Windows XP. Furthermore, applications using Wull6 must be run with administrator rights.

---

[8] Cf. WULL6Startup definition (6.19).

5. Addressing

Users must pay attention to IPv6 addressing/routing. In an IPv4 environment, knowing which source address to use when constructing a packet is easy, because there's only one IPv4 address per interface.

In an IPv6 environment, several IPv6 addresses can be linked to a same interface. Wull6 must find the right address source to use, depending on the information about the destination and the local addresses.

The matching algorithm Wull6 is based on the longest prefix: before sending (the same logic is adopted is for connecting), Wull6 calculate which is the local address that has the longest common prefix part with the destination and uses it as source address.

6. Wull6 API

6.1. WULL66Bind

The **WULL66Bind** function associates a local address with a socket.

```
int WULL66Bind(

  SOCKET s,

  const struct sockaddr* name,

  int namelen

);
```

Parameters

s

[in] Descriptor identifying an unbound socket.

Name

[in] Address to assign to the socket from the sockaddr structure.

namelen

[in] Length of the value in the *name* parameter, in bytes.

Return Values

If no error occurs, **WULL66Bind** returns zero. Otherwise, it returns **SOCKET_ERROR**, and a specific error code can be retrieved by calling **WULL66GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL66Startup** call must occur before using this function.
WSAEADDRINUSE	A process on the computer is already bound to the same fully-qualified address and the socket has not been marked to allow address reuse with SO_REUSEADDR. For example, the IP address and port are bound in the af_inet case). (See the SO_REUSEADDR socket option under **WULL66setsockopt**.)
WSAEADDRNOTAVAIL	The specified address is not a valid address for this computer.
WSAEINVAL	The socket is already bound to an address.

WSAENOTSOCK	The descriptor is not a socket.

6.2. WULL66Cleanup

The **WULL66Cleanup** function terminates use of the Wull6.

```
int WULL66Cleanup(void);
```

Parameters

This function has no parameters.

Return Values

The return value is zero if the operation was successful. Otherwise, the value **SOCKET_ERROR** is returned, and a specific error number can be retrieved by calling **WULL66GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL66Startup** call must occur before using this function.

6.3. WULL66CloseEvent

The **WULL66CloseEvent** function closes an open event object handle.

```
BOOL WULL66CloseEvent(

  WSAEVENT hEvent

);
```

Parameters

hEvent

[in] Object handle identifying the open event.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **WULL66GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL66Startup** call must occur before using this function.

WSA_INVALID_HANDLE	The *hEvent* is not a valid event object handle.

Remarks

The handle to the event object is closed so that further references to this handle will fail with the error WSA_INVALID_HANDLE.

6.4. WULL66CloseSocket

The **WULL66CloseSocket** function closes an existing socket.

```
int WULL66CloseSocket(

  SOCKET s

);
```

Parameters

s

[in] Descriptor identifying the socket to close.

Return Values

If no error occurs, **WULL66CloseSocket** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WULL66GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL66Startup** call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.

6.5. WULL66Connect

The **WULL6Connect** function establishes a connection to a specified socket.

```
int WULL6Connect(

  SOCKET s,

  const struct sockaddr* name,

  int namelen
```

```
);
```

Parameters

s

[in] Descriptor identifying an unconnected socket.

name

[in] Name of the socket in the **sockaddr** structure to which the connection should be established.

namelen

[in] Length of *name*, in bytes

Return Values

If no error occurs, **WULL6Connect** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code can be retrieved by calling **WULL6GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSAEADDRNOTAVAIL	The specified address is not a valid address for this computer.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket
WSAEINVAL	The socket is already bound to an address.
WSAENOTSOCK	The descriptor is not a socket.

6.6. WULL6CreateEvent

The **WULL6CreateEvent** function creates a new event object.

```
HANDLE WULL6CreateEvent(void);
```

Parameters

This function has no parameters.

Return Values

If no error occurs, **WULL6CreateEvent** returns the handle of the event object. Otherwise, the return value is WSA_INVALID_EVENT. To get extended error information, call **WULL6GetLastError**.

Error code	Meaning

WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to create the event object

6.7. WULL6EventSelect

The **WULL6EventSelect** function specifies an event object to be associated with the FD_READ network event.

```
int WULL6EventSelect(

  SOCKET s,

  HANDLE hEventObject,

  long lNetworkEvents

);
```

Parameters

s

[in] Descriptor identifying the socket.

hEventObject

[in] Handle identifying the event object to be associated the FD_READ network event.

lNetworkEvents

[in] FD_READ network event or 0 to set the socket in blocking mode again.

Return Values

The return value is zero if the application's specification of the network events and the associated event object was successful. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling **WULL6GetLastError**.

6.8. WULL6GetLastError

The **WULL6GetLastError** function returns the error status for the last operation that failed.

```
int WULL6GetLastError(void);
```

Parameters

This function has no parameters.

Return Values

The return value indicates the error code for this thread's last WULL6 Sockets operation that failed.

6.9. WULL6getsockname

The **WULL6getsockname** function retrieves the local name for a socket.

```
int WULL6getsockname(

  SOCKET s,

  struct sockaddr* name,

  int* namelen

);
```

Parameters

s

[in] Descriptor identifying a socket.

name

[out] Pointer to a **SOCKADDR** structure that receives the address (name) of the socket.

namelen

[in, out] Size of the *name* buffer, in bytes.

Return Values

If no error occurs, **WULL6getsockname** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WULL6GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound to an address with **WULL6bind**, or ADDR_ANY is specified in **WULL6bind** but connection has not yet occurs.

6.10. WULL6getsockopt

The **WULL6getsockopt** function only retrieves the checksum coverage value.

```
int WULL6getsockopt(

  SOCKET s,

  int level,

  int optname,

  char* optval,

  int* optlen

);
```

Parameters

s

[in] Descriptor identifying a socket.

level

[in] Level at which the option is defined; the supported levels are IPPROTO_UDPLITE and IPPROTO_IP.

optname

[in] Socket option for which the value is to be set. The supported options are UDPLITE_CHECKSUM_COVERAGE for IPPROTO_UDPLITE level and IP_TTL for IPPROTO_IP .

optval

[out] Pointer to the buffer in which the value for the requested option is to be returned.

optlen

[in, out] Pointer to the size of the *optval* buffer, in bytes.

Return Values

If no error occurs, **WULL6setsockopt** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WULL6GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The *level* parameter is unknown or invalid.
WSAENOPROTOOPT	The option is unknown or unsupported by the indicated protocol family.

6.11. WULL6ioctlsocket

The **WULL6ioctlsocket** function controls the I/O mode of a socket.

```
int WULL6ioctlsocket (

  SOCKET s,

  long cmd,

  u_long* argp

);
```

Parameters

s

[in] Descriptor identifying a socket.

cmd

[in] Command to perform on the socket *s*.

argp

[in, out] Pointer to a parameter for *cmd*.

Return Values

Upon successful completion, the **WULL6ioctlsocket** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WULL6GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINPROGRESS	It is not possible to change the state of the socket because it is already used with an event (a call to WULL6EventSelect has been made).

Remarks

The **WULL6ioctlsocket** function can be used on any socket in any state. It is used to set or retrieve operating parameters associated with the socket, independent of the protocol and communications subsystem. Here is the supported command to use in the *cmd* parameter and its semantic:

FIONBIO

The *argp* parameter is a pointer to an unsigned long value. Set *argp* to a nonzero value if the nonblocking mode should be enabled, or zero if the nonblocking mode should be disabled. When a socket is created, it operates in blocking mode by default (nonblocking mode is disabled). This is consistent with BSD sockets.

argp value	Nonblocking mode
0	Disabled
nonzero	Enabled

6.12. WULL6Recv

The **WULL6Recv** function receives data from a connected or bound socket.

```
int WULL6Recv (

  SOCKET s,

  char* buf,
```

```
  int len,

  int flags

);
```

Parameters

s

[in] Descriptor identifying a connected socket.

buf

[out] Buffer for the incoming data.

len

[in] Length of *buf*, in bytes

flags

[in] Flag specifying the way in which the call is made. Should be 0 (flag not used in WULL6).

Return Values

If no error occurs, **WULL6Recv** returns the number of bytes received. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WULL6GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound with **WULL6Bind**
WSAENOTCONN	The socket is not connected.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.

Remarks

The **WULL6Recv** function is used to read incoming data on connection-oriented sockets, or connectionless sockets. The WULL66 sockets must be bound before calling **WULL6Recv**.

The local address of the socket must be known. For server applications, use an explicit **WULL6Bind**. Explicit binding is discouraged for client applications. For client applications, the socket can become bound implicitly to a local address using **WULL6Connect**.

For connected or connectionless sockets, the **WULL6Recv** function restricts the addresses from which received messages are accepted. The function only returns messages from the remote address specified in the connection. Messages from other addresses are discarded.

For connectionless sockets (type SOCK_DGRAM or other message-oriented sockets), data is extracted from the first enqueued datagram (message) from the destination address specified by the **WULL6Connect** function.

If the datagram or message is larger than the buffer specified, the buffer is filled with the first part of the datagram, and **WULL6Recv** generates the error **WSAEMSGSIZE**. The excess data is lost.If no incoming data is available at the socket, the **WULL6Recv** call blocks and waits for data to arrive unless the socket is nonblocking.

6.13. WULL6RecvFrom

The **WULL6RecvFrom** function receives a datagram and stores the source address.

```
int WULL6RecvFrom (

  SOCKET s,

  char* buf,

  int len,

  int flags,

  struct sockaddr* from,

  int* fromlen

);
```

Parameters

s

[in] Descriptor identifying a bound socket.

buf

[out] Buffer for the incoming data.

len

[in] Length of *buf*, in bytes.

flags

[in] Indicator specifying the way in which the call is made. Should be 0 (flag not used in WULL6).

from

[out] Optional pointer to a buffer in a **sockaddr** structure that will hold the source address upon return.

fromlen

[in, out] Optional pointer to the size, in bytes, of the *from* buffer.

Return Values

If no error occurs, **WULL6RecvFrom** returns the number of bytes received. If **SOCKET_ERROR** is returned, a specific error code can be retrieved by calling **WULL6GetLastError**.

The number of bytes received corresponds to the number of bytes of the UDP-Lite payload.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound with **WULL6Bind**.
WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection oriented or connectionless.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAEFAULT	The *buf* or *from* parameters are not part of the user address space, or the *fromlen* parameter is too small to accommodate the peer address.
WULL6SENDBADCHECKSUM	The checksum coverage configured for this socket is greater than the size of packet to send; it is not possible to send this packet (or it will be discarded at the end point).

Remarks

The **WULL6RecvFrom** function reads incoming data on both connected and unconnected sockets and captures the address from which the data was sent. The socket must not be connected. The local address of the socket must be known. For server applications, this is usually done explicitly through **WULL6Bind**.

If the *from* parameter is nonzero and the socket is not connection oriented, (type SOCK_DGRAM for example), the network address of the peer that sent the data is copied to the corresponding **sockaddr** structure. The value pointed to by *fromlen* is initialized to the size of this structure and is modified, on return, to indicate the actual size of the address stored in the **sockaddr** structure.

If no incoming data is available at the socket, the **WULL6RecvFrom** function blocks and waits for data to arrive set unless the socket is nonblocking. **WULL6EventSelect** can be used to determine when more data arrives.

6.14. WULL6ResetEvent

The **WULL6ResetEvent** function resets the state of the specified event object to nonsignaled.

```
BOOL WULL6ResetEvent(

  WSAEVENT hEvent

);
```

Parameters

hEvent

[in] Handle that identifies an open event object handle.

Return Values

If the **WULL6ResetEvent** function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call **WULL6GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSA_INVALID_HANDLE	The *hEvent* parameter is not a valid event object handle.

Remarks

The **WULL6ResetEvent** function is used to set the state of the event object to non signaled.

6.15. WULL6Send

The **WULL6Send** function sends data on a connected socket.

```
int WULL6Send (
```

```
  SOCKET s,

  const char* buf,

  int len,

  int flags

);
```

Parameters

s

[in] Descriptor identifying a connected socket.

buf

[in] Buffer containing the data to be transmitted.

len

[in] Length of the data in *buf*, in bytes

flags

[in] Indicator specifying the way in which the call is made. Should be 0 (flag not used in WULL6).

Return Values

If no error occurs, **WULL6Send** returns the total number of bytes sent, which can be less than the number indicated by *len*. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WULL6GetLastError**.

The number of bytes sent corresponds to the number of bytes of the UDPLite payload.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound with **WULL6Bind**.
WSAENOTCONN	The socket is not connected.
WULL6SENDBADCHECKSUM	The checksum coverage configured for this socket is greater than the size of packet to send; it is not possible to send this packet (or it will be discarded at the end point).

WULL6SENDBUFFERTOOSHORT	The data length to send is too big for the sending buffer .

Remarks

The **WULL6Send** function is used to write outgoing data on a connected socket.

The successful completion of a **WULL6Send** does not indicate that the data was successfully delivered.

Calling **WULL6Send** with a zero *len* parameter is permissible and will be treated by implementations as successful. In such cases, **WULL6Send** will return zero as a valid value. For message-oriented sockets, a zero-length transport datagram is sent.

6.16. WULL6SendTo

The **WULL6SendTo** function sends data to a specific destination.

```
int WULL6SendTO(

  SOCKET s,

  const char* buf,

  int len,

  int flags,

  const struct sockaddr* to,

  int tolen

);
```

Parameters

s

[in] Descriptor identifying a (possibly connected) WULL6 socket.

buf

[in] Buffer containing the data to be transmitted.

len

[in] Length of the data in *buf*, in bytes.

flags

[in] Indicator specifying the way in which the call is made. Should be 0 (flag not used in WULL6).

to

[in] Optional pointer to a **sockaddr** structure that contains the address of the target socket.

tolen

[in] Size of the address in *to*, in bytes.

Return Values

If no error occurs, **WULL6SendTo** returns the total number of bytes sent, which can be less than the number indicated by *len*. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WULL6GetLastError**.

The number of bytes sent corresponds to the number of bytes of the UDPLite payload.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEDESTADDRREQ	A destination address is required.
WSAEFAULT	The *buf* or *to* parameters are not part of the user address space, or the *tolen* parameter is too small.
WULL6SENDBADCHECKSUM	The checksum coverage configured for this socket is greater than the size of packet to send; it is not possible to send this packet (or it will be discarded at the end point).
WULL6SENDBUFFERTOOSHORT	The data length to send is too big for the sending buffer .

Remarks

The **WULL6SendTo** function is used to write outgoing data on a socket.

The successful completion of a **WULL6SendTo** does not indicate that the data was successfully delivered.

The **WULL6SendTo** function is normally used on a connectionless socket to send a datagram to a specific peer socket identified by the *to* parameter. Even if the connectionless socket has been previously connected to a specific address, the *to* parameter overrides the destination address for that particular datagram only.

6.17. WULL6setsockopt

The **WULL6setsockopt** function only sets the checksum coverage socket option.

```
int WULL6setsockopt (

  SOCKET s,

  int level,

  int optname,

  const char* optval,

  int optlen

);
```

Parameters

s

[in] Descriptor identifying a socket.

level

[in] Level at which the option is defined; the supported levels are IPPROTO_UDPLITE and IPPROTO_IP.

optname

[in] Socket option for which the value is to be set. The supported options are UDPLITE_CHECKSUM_COVERAGE for IPPROTO_UDPLITE level and IP_TTL for IPPROTO_IP .Default ttl value is 32.

optval

[in] Pointer to the buffer in which the value for the requested option is specified.

optlen

[in] Size of the *optval* buffer, in bytes.

Return Values

If no error occurs, **WULL6setsockopt** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WULL6GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.
WSAEINVAL	*level* is not valid, or the information in *optval* is not valid.
WSAENOTSOCK	The descriptor is not a socket.

6.18. WULL6Socket

The **WULL6Socket** function creates a socket that is bound to a specific transport-service provider. The only transport protocol authorized is IPROTO_UDPLITE.

```
SOCKET WULL6Socket(

  int af,

  int type,

  int protocol,

  );
```

Parameters

af

[in] Address family specification.

type

[in] Type specification for the new socket.

protocol

[in] Protocol to be used with the socket that is specific to the indicated address family.

Return Values

If no error occurs, **WULL6Socket** returns a descriptor referencing the new socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code can be retrieved by calling **WULL6GetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WULL6Startup** call must occur before using this function.

WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEMFILE	No more socket descriptors are available.
WSAEPROTOTYPE	The proto type is not supported (should be SOCK_DGRAM)
WSAEPROTONOSUPPORT	The specified protocol is not supported (must be IPPROTO_UDPLITE).

Remarks

The **WULSocket** function causes a socket descriptor and any related resources to be allocated and associated with a transport-service provider. The only supported protocol is IPPROTO_UDPLITE and the only supported address family specification is AF_INET.

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using **WULL6SendTo** and **WULL6RecvFrom**. If such a socket is connected to a specific peer, datagrams can be sent to that peer using **WULL6Send** and can be received from (only) this peer using **WULL6Recv**.

6.19. WULL6Startup

The **WULL6Startup** function initiates use of Wull6.DLL by a process and can enable a possible error bit generating function.

```
int WULL6Startup(int (*f)(char* buffer, unsigned int nBuffLen)=NULL);
```

Parameters

f

[in] Pointer to an error bit generating function, which is called once IP/UDP-Lite packet are ready to send in order to introduce bit errors. Wull does not provide developers with such functions. They have to write them[9].

Return Values

The **WULL6Startup** function returns zero if successful. Otherwise, it returns one of the error codes listed in the following.

An application cannot call **WULL6GetLastError** to determine the error code as is normally done in Windows Sockets if **WullStartup** fails. The wull.dll will *not* have been loaded in the case of a failure so the client data area where the last error information is stored could not be established.

Error code	Meaning

[9] An example function is presented in Annexe A.

WULL6NOKEY	No key has been attributed for this dll. It means that Wull setup has not been properly mase and no license is accorded to wull dll.
WULL6INVALIDKEY	The Wull key is invalid: it doesn't have the correct format or limit date has been exceeded. You should ask for a new key .
WSASYSNOTREADY	Indicates that the underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	The version of Windows Sockets support requested is not provided by this particular Windows Sockets implementation.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 operation is in progress.
WSAEPROCLIM	Limit on the number of tasks supported by the Windows Sockets implementation has been reached.
WSAEFAULT	The call to WSAStartup has failed.

The 5 last error codes are the result of a call to WSAStartup, used in WULL6.

7. Installation and configuration

Wull6 must be installed and configured to make use of it. The following chapters detail these two steps.

7.1. Installation

Wull6 is shipped with a setup utility (“Wull6Setup.exe”) that partners have to use to enter the software key they have been provided with.

A specific 20 byte character long key is generated for every Wull6 partners. Thanks to this key, applications using Wull6 can still run up to a limited date. Once a key is out-of-date feel free to ask a new key: they are free of charge.

To make the DLL valid, just execute “Wull6Setup.exe” and enter your dedicated key:

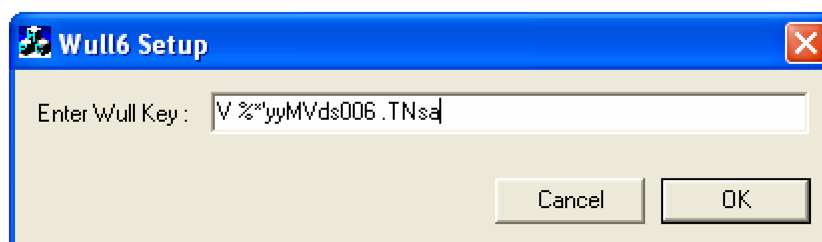

Figure 4: Entering the software key

7.2. Configuration

Once Wull6 is installed, it can run without further configuration. However, Wull6 has been designed to make its use as easy as possible whatever your configuration.

One default parameters can be modified: the protocol number dedicated to UDP-Lite.

The very new RFC 3828 describes UDP-Lite as a proposed standard. The proposed protocol number is 136. However, Wull can use a different protocol number if needed.

To do so, you must have administrator rights and execute the “regedit” command. Then set the “HKEY_LOCAL_MACHINES\SOFTWARES\Irisa\Wull\ProtocolNumber” to the selected value.

To make your program compliant with this protocol value, update it in the “wull.h” file and compile your program again.

7.3. Restriction of use

Here is a list summarizing Wull6’s restrictions of use:

- Wull6 only runs on Windows 2000 and XP
- Only IPv6 in unicast mode is supported
- Wull6 must be run with administrator rights
- The maximal number of sockets that Wull6 can handle is set to 30.
- WULL66 will perfectly run only on single-homed computers.

Annexe A A full receiver example

The following source code is a receiver application that transfers using UDP-Lite. The receiver takes advantage of Windows events to be notified when packets arrive (cf. WaitForMultipleObjects call).

- **Receiver**: r <file name> <IP address- useless as receiver> < Port number> <bitrate useless as receiver>

Source code :

```
#include "stdafx.h"

#include <stdio.h>
#include <stdlib.h>
#include <Afxmt.h>
#include <afxdisp.h>
#include	<winsock2.h>
#include	<ws2tcpip.h>
#include	<winsock2.h>
#include	<ws2tcpip.h>

#include <tpipv6.h>
#include "wull6.h"


#define			SIZE_BUF	1400

	/**
	*
	* @param 1 as filename to receive
	* @param 2 as receiver1/sender
	* @param 3 as port
	*/
int main(int argc, char* argv[])
{
	FILE*		pFile3 = NULL;
	FILE*		pFile1 = NULL;
	FILE*		pFile2 = NULL;
	SOCKET		sMySocket = INVALID_SOCKET;
	bool		bSender = true;
	int		nRetour=-1;
	int		nPort;
	int		nRet = 0;
	int		nNbOfBytesToSend = 0;
	int		nNbMaxOfBytesToSend = 0;
```

```
char            sBuffer[1024];
SOCKADDR_IN     ReceiverAddr1;
HANDLE          npmLect;

DWORD           dWaitValue;

SOCKADDR_IN LocalAddr;
long            nlFileLength1 = 0;
long            nlFileLength2 = 0;
long            nlFileLength3 = 0;
long            nlReceivedLength = 0;
long            nlSentLength1 = 0;
long            nlSentLength2 = 0;
long            nlSentLength3 = 0;
int             nSleepDuration = 0;
int             nTotalSent1 = 0;
int             nTotalSent2 = 0;
int             nTotalSent3 = 0;
bool            bIAmDead=false;
int             nReadBytes;
bool            bDead1=false;
bool            bDead2=false;
bool            bInitialise=false;
char            ReceiveBuf[SIZE_BUF];
long            nlFileLength = 0;


if(argc!=4)
{
        printf("\nCall: <filename> <receiver> <port>  ");
        return -1;
}
nPort = atoi(argv[3]);
HANDLE          aEventHandle[WSA_MAXIMUM_WAIT_EVENTS];
int     nNbOfEvents;


if ((nRet = WULL6Startup()) != 0)
{
        // NOTE: Since Winsock failed to load we cannot use WSAGetLastError
        // to determine the error code as is normally done when a Winsock
        // API fails. We have to report the return status of the function.
        return 0;
}
WSADATA wsaData;
//get the local address
if ((nRetour = WSAStartup(MAKEWORD(2,2), &wsaData)) != 0)
{
```

```
		// NOTE: Since Winsock failed to load we cannot use
		// WSAGetLastError to determine the specific error for
		// why it failed. Instead we can rely on the return
		// status of WSAStartup.
		printf("WSAStartup failed with error %d\n", nRetour);
	}

	/*	get the strcuture info for destination  address */
	struct addrinfo aiHints;
	struct addrinfo *aiList = NULL;
	 memset(&aiHints, 0, sizeof(aiHints));
	aiHints.ai_family = AF_INET6;
	//very important
	aiHints.ai_flags = AI_NUMERICHOST | AI_PASSIVE;

	if ((nRetour = getaddrinfo(argv[2],argv[3], &aiHints, &aiList)) != 0)
	{
		printf("prog principal getaddrinfo() failed.\n");}
	else
	{
		printf ("\n protocol %d",aiList->ai_protocol);
		printf ("\n socket type %d",aiList->ai_socktype);
		printf ("\n address size %d",aiList->ai_addrlen);
		printf ("\n address family %d",aiList->ai_addr->sa_family);
		printf ("\n address data %d",aiList->ai_addr->sa_data);

		if(aiList->ai_next!=NULL)
		{
			aiList=aiList->ai_next;
			printf ("\n protocol %d",aiList->ai_protocol);
			printf ("\n socket type %d",aiList->ai_socktype);
			printf ("\n address size %d",aiList->ai_addrlen);
			printf ("\n address family %d",aiList->ai_addr->sa_family);
			printf ("\n address data %d",aiList->ai_addr->sa_data);
			printf ("\n local name %s",aiList->ai_canonname);
		}
	}
	char sAdresse[50];
	nRetour=getnameinfo(aiList->ai_addr,aiList-
>ai_addrlen,sAdresse,sizeof(sAdresse),NULL,0,NI_NUMERICHOST );
	if(nRetour<0)
	{
		printf("\n ERREUR getnameinfo");
	}
	else{
		printf("Hote distant %s",sAdresse);
	}
```

```
 nPort=atoi(argv[3]);
SOCKADDR_IN6 Ici;
Ici.sin6_family=AF_INET6;
Ici.sin6_addr=in6addr_any;
Ici.sin6_port=htons(nPort);

sMySocket = WULL6Socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDPLITE);
if (sMySocket == INVALID_SOCKET)
{
	printf("ERROR: socket failed with error %d\n", WULL6GetLastError());
}

printf("\nNumero de la socket %d",sMySocket);

/**Bind des sockets*/
nRetour = WULL6Bind(sMySocket, (SOCKADDR*)&Ici,sizeof(Ici));
if (nRetour<0)
{
	printf("ERROR: socket BIND failed with error %d\n",
WULL6GetLastError());
}
printf("\nBind OK");

nRetour= WULL6Connect(sMySocket,aiList->ai_addr,aiList->ai_addrlen);
if (nRetour<0)
{
	printf("ErreurConnect %d",WULL6GetLastError());
}
printf("\nConnect OK");

/**envoi de la taille*/
COleDateTime timeStart;
COleDateTime timeStop;

if ((nRet = WULL6Recv(sMySocket, ReceiveBuf, sizeof(int), 0)) ==
SOCKET_ERROR)
{
	WULL6CloseSocket(sMySocket);
	return 0;
}
if ((nRet = WULL6Send(sMySocket, ReceiveBuf, sizeof(int), 0)) ==
SOCKET_ERROR)
{
	WULL6CloseSocket(sMySocket);
	return 0;
}
```

```
	printf("\nlongueur reçue ");
	timeStart = COleDateTime::GetCurrentTime();
	nlFileLength = *((long*) ReceiveBuf);
	printf("\nReception de lalongueur %d, début de la boucle ",nlFileLength);

	npmLect=WULL6CreateEvent();
	nRetour= WULL6EventSelect(sMySocket,npmLect,FD_READ);
	if(nRetour!=0)
	{
		printf("erreur NPMSelect %d",WULL6GetLastError());
	}

	// link that event to the previously created socket
	aEventHandle[0]=npmLect;

	pFile1 = fopen(argv[1], "wb");
	if (!pFile1)
	{
		printf("Reception file not created");
		return 0;
	}

	nNbOfEvents=1;
	int nReceivedBytes=0;

	while (!bIAmDead)
	{
		dWaitValue = WaitForMultipleObjects(nNbOfEvents, aEventHandle,
	FALSE, INFINITE);
#ifdef _DEBUG
		if (dWaitValue == WSA_WAIT_FAILED)
		{
			int  nLastError = WULL6GetLastError ();
			ASSERT(0);
		}
#endif

		// an event is signaled
		int nIndex = dWaitValue - WAIT_OBJECT_0;
		ASSERT(nIndex >= 0);
		switch(nIndex)
		{
		case 0: //arrivée de données sur la socket
			if(nlReceivedLength < nlFileLength)
			{
				// read data
```

```
				if ((nReceivedBytes = WULL6Recv(sMySocket, ReceiveBuf,
sizeof(ReceiveBuf), 0)) == SOCKET_ERROR)
				{
					closesocket(sMySocket);
					return 0;
				}
				nlReceivedLength += nRet;
				// write data into the file
				nRet=fwrite(ReceiveBuf, nReceivedBytes, sizeof(char), pFile1);
				if(nRet<0)
				{
					printf("\nErreur fwrite");
				}
				printf("received bytes : %d\n", nlReceivedLength);
				if ((nRet = WULL6Send(sMySocket, ReceiveBuf,
nReceivedBytes, 0)) == SOCKET_ERROR)
				{
					closesocket(sMySocket);
					return 0;
				}
				if (nlReceivedLength==nlFileLength)
				{
					bIAmDead=true;
				}
			}
			else
			{
				bIAmDead=true;
			}
			break;

		}//fin du switch
	}//Fin du while*/

	//Fermeture de toutes les sockets
	WULL6CloseSocket( sMySocket);
	WULL6Cleanup();
	return 0;
}
```

Annexe B Error codes

Return codes of WSASockets have been reused for WULL66, in order to make the implementation of application using both types of sockets easier. These errors are accessible thanks to the WULL6GetLastError function.

The following is a list of possible error codes returned by the **WULL6GetLastError** call, along with their extended explanations. Errors are listed in alphabetical order by error macro. Some error codes defined in WINSOCK2.H are not returned from any function - these have not been listed here.

WSAEADDRINUSE

(10048)

Address already in use.

Only one usage of each socket address (protocol/IP address/port) is normally permitted. This error occurs if an application attempts to **WULL6Bind** a socket to an IP address/port that has already been used for an existing socket, or a socket that wasn't closed properly, or one that is still in the process of closing. For server applications that need to **bind** multiple sockets to the same port number, consider using **WULL6setsockopt(SO_REUSEADDR)**. Client applications usually need not call **WULL6Bind** at all - **WULL6Connect** will choose an unused port automatically. When **WULL6Bind** is called with a wild-card address (involving ADDR_ANY), a WSAEADDRINUSE error could be delayed until the specific address is "committed." This could happen with a call to other function later, including **WULL6Connect**.

WSAEADDRNOTAVAIL

(10049)

Cannot assign requested address.

The requested address is not valid in its context. Normally results from an attempt to **WULL6Bind** to an address that is not valid for the local machine. This can also result from **WULL6Connect**, **WULL6SendTo** when the remote address or port is not valid for a remote machine (e.g. address or port 0).

WSAEAFNOSUPPORT

(10047)

Address family not supported by protocol family.

An address incompatible with the requested protocol was used. All sockets are created with an associated "address family" (i.e. AF_INET for Internet Protocols) and a generic protocol type (i.e. SOCK_STREAM). This error will be returned if an

incorrect protocol is explicitly requested in the **WULL6Socket** call, or if an address of the wrong family is used for a socket, e.g. in WULSendTo.

WSAEDESTADDRREQ

(10039)

Destination address required.

A required address was omitted from an operation on a socket. For example, this error will be returned if **WULL6SendTo** is called with the remote address of ADDR_ANY.

WSAEINVAL

(10022)

Invalid argument.

Some invalid argument was supplied (for example, specifying an invalid level to the **WULL6setsockopt** function).

WSAEISCONN

(10056)

Socket is already connected.

A connect request was made on an already connected socket. Some implementations also return this error if **WULL6SendTo** is called on a connected SOCK_DGRAM socket (For SOCK_STREAM sockets, the *to* parameter in **WULL6SendTo** is ignored), although other implementations treat this as a legal occurrence.

WSAEMFILE

(10024)

Too many open files.

Too many open sockets. Each implementation may have a maximum number of socket handles available, either globally, per process or per thread.

WSAEMSGSIZE

(10040)

Message too long.

A message sent on a datagram socket was larger than the internal message buffer or some other network limit, or the buffer used to receive a datagram into was smaller than the datagram itself.

WSAENOPROTOOPT

(10042)

Bad protocol option.

An unknown, invalid or unsupported option or level was specified in a **WULL6getsockopt** or **WULL6setsockopt** call.

WSAENOTCONN

(10057)

Socket is not connected.

A request to send or receive data was disallowed because the socket is not connected and (when sending on a datagram socket using **WULL6SendTo**) no address was supplied. Any other type of operation might also return this error - for example, **WULL6setsockopt** setting SO_KEEPALIVE if the connection has been reset.

WSAEINPROGRESS

(10036)

A blocking operation is currently executing.

WSAENOTSOCK

(10038)

Socket operation on non-socket.

An operation was attempted on something that is not a socket. The socket handle parameter did not reference a valid socket.

WSAEPROTONOSUPPORT

(10043)

Protocol not supported.

The requested protocol has not been configured into the system, or no implementation for it exists. For example, a **WULL6Socket** call requests a SOCK_DGRAM socket, but specifies a stream protocol.

WSAEPROTOTYPE

(10041)

Protocol wrong type for socket.

A protocol was specified in the **WULL6Socket** function call that does not support the semantics of the socket type requested. For example, the ARPA Internet UDP protocol cannot be specified with a socket type of SOCK_STREAM.

WSANOTINITIALISED

(10093)

Successful WSAStartup not yet performed.

Either the application hasn't called **WULL6Startup** or **WULL6Startup** failed. The application may be accessing a socket which the current active task does not own (i.e. trying to share a socket between tasks), or **WULL6Cleanup** has been called too many times.

WULL6NOKEY

(10240)

No key has been attributed for this dll. It means that Wull6 setup has not been properly mase and no license is accorded to wull6 dll.

WULL6INVALIDKEY

(10241)

The Wull6 key is invalid: it doesn't have the correct format or limit date has been exceeded. You should ask for a new key .

WULL6SENDBADCHECKSUM

(10242)

The checksum coverage configured for this socket is greater than the size of packet to send; it is not possible to send this packet (or it will be discarded at the end point).

WULL6SENDBUFFERTOOSHORT

(10243)

The data length to send is too big for the sending buffer .