



institut de recherche en informatique
et systèmes aléatoires

WULL: A WINDOWS UDP-LITE LIBRARY

USER'S GUIDE

Version: 1.0.0.9

Date : 10 Jan. 05.

Auteur(s):

Laurent Guillo

Cécile Marc



Table of contents

1. WHAT IS WULL?	6
2. UDP-LITE A QUICK OVERVIEW	8
3. A FIRST EXAMPLE	10
4. OVERALL ARCHITECTURE	12
5. WULL API	14
5.1. WULLBIND	14
5.2. WULLCLEANUP	14
5.3. WULLCLOSEEVENT	15
5.4. WULLCLOSESOCKET	15
5.5. WULLCONNECT	16
5.6. WULLCREATEEVENT	17
5.7. WULLEVENTSELECT	17
5.8. WULLGETLASTERROR	18
5.9. WULLGETSOCKNAME	18
5.10. WULLGETSOCKOPT	19
5.11. WULLIOCTLSOCKET	20
5.12. WULLRECV	21
5.13. WULLRECVFROM	22
5.14. WULLRESETEVENT	24
5.15. WULLSEND	24
5.16. WULLSENDTO	25
5.17. WULLSETSOCKOPT	27
5.18. WULLSOCKET	28
5.19. WULLSTARTUP	29
5.20. WULLVIRTUALSEND	30
5.21. WULLVIRTUALSENDTO	31
6. INSTALLATION AND CONFIGURATION	35
6.1. INSTALLATION	35
6.2. CONFIGURATION	35
6.2.1. <i>Network interface</i>	35
6.2.2. <i>Protocol number</i>	35
6.3. RESTRICTION OF USE	36
ANNEXE A PERFORMANCE: A FEW FIGURES	37
ANNEXE B A FULL SENDER/RECEIVER EXAMPLE	38
ANNEXE C ERROR CODES	43

1. What is Wull?

Wull¹ is Windows DLL², which implements the UDP-Lite protocol for IPv4 in unicast mode only. Wull 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 Wull 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 Wull API.
- **General architecture** outlines how Wull implements UDP lite.
- **Wull API** describes all functions Wull provides developers with.
- **Installation and configuration** lists all the required steps to install and to set Wull.

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.

Since July 2004, UDP-Lite is described in the standard RFC 3828³. Wull is compliant with that RFC.

¹ Wull : UDP lite library.

² Dynamic-linked library.

³ Les RFC sont accessibles sur le site de l'IETF: www.ietf.org.

2. UDP-lite a quick overview

There is a 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 packet⁴, 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:

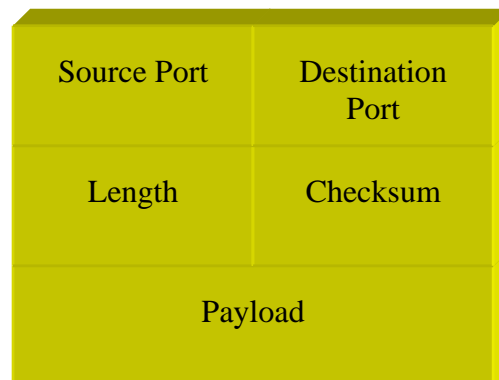


Figure 1: UDP packet format

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:

- 32 bit IP source address field,
- 32 bit IP source destination field,
- 8 bit zero field,
- 8 bit protocol field,
- 16 bit UDP length field.

⁴ which is Wull's default behaviour.

A UDP-Lite packet has the following format:

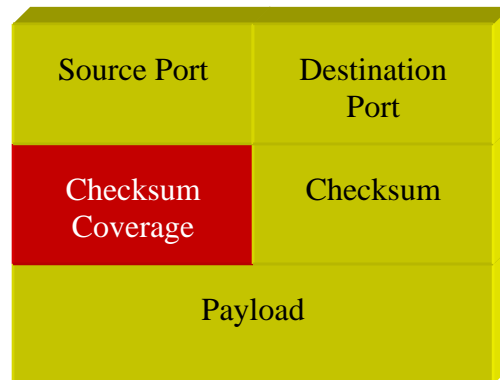


Figure 2: UDP-Lite packet format

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. The Wull API calls are in bold type.

```
if ((nRet = WULLStartup() != 0))
{
    printf("ERROR: WULLStartup failed with error %d\n", nRet);
    return 0;
}
// create a UDP lite socket
sSocketTest = WULLSocket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE);
if (sSocketTest == INVALID_SOCKET)
{
    printf("ERROR: socket failed with error %d\n", WULLGetLastError());
    WULLCleanup();
    return 0;
}
// Setup a SOCKADDR_IN structure that will identify who we will send
datagrams to.
ReceiverAddr.sin_family = AF_INET;
ReceiverAddr.sin_port = htons(6970);
ReceiverAddr.sin_addr.s_addr = inet_addr("168.254.70.76");

// set the checksum coverage
int optchecksum=8;
if(WULLsetsockopt(sMySocket, IPPROTO_UDPLITE, UDPLITE_CHECKSUM_COVERAGE, (char
*)&optchecksum, sizeof(optchecksum) < 0){
    printf("\nerreur setsockopt() %d", WULLGetLastError());
}
// Send a datagram to the receiver.
if ((nRet = WULLSendTo(sSocketTest, SendBuf, sizeof(SendBuf), 0, (SOCKADDR
*)&ReceiverAddr, sizeof(ReceiverAddr))) == SOCKET_ERROR){
    printf("ERROR: sendto failed with error %d\n", WULLGetLastError());
}
// When your application is finished sending datagrams close the socket.
WULLCloseSocket(sSocketTest);
// When your application is finished call WULLCleanup.
WULLCleanup();
```

This example outlines how to:

- Initialize the Wull API,
- Create a UDP-Lite socket,
- Set the checksum coverage,
- Send data,
- Stop the Wull Interface.

The required steps are now described. First, a Windows developer has already noticed that functions belonging to the Wull API start with the string “WULL”. Moreover, they are similar to Windows socket API functions.

The first step is to initialize the Wull API with the “WULLStartup” function. Without this call, all following calls to the other Wull functions will fail⁵.

Wull allows developers to create sockets with UDP-Lite as protocol. To do so, a new value for the “WULLSocket”’s protocol argument is added: IPPROTO_UDPLITE.

To specify the checksum coverage, Wull takes advantage of the “WULLsetsockopt” 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 “WULLsetsockopt” function.

Sending a packet is similar to the Windows “classic” socket way. The function “WULLSendTo” used in the example or the other sending function “WULLSend” are analogous to Windows’ ones.

As the Wull API has been initialized, it must be properly stopped. That is why the application calls “WULLCleanup”.

All functions of the Wull API can fail. To get more information about why, call the “WULLGetLastError” function⁶.

⁵ WULLStartup may return an error, cf. 5.1.

⁶ Error codes are listed in Annexe C.

4. Overall architecture

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

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

The following diagram describes the overall architecture:

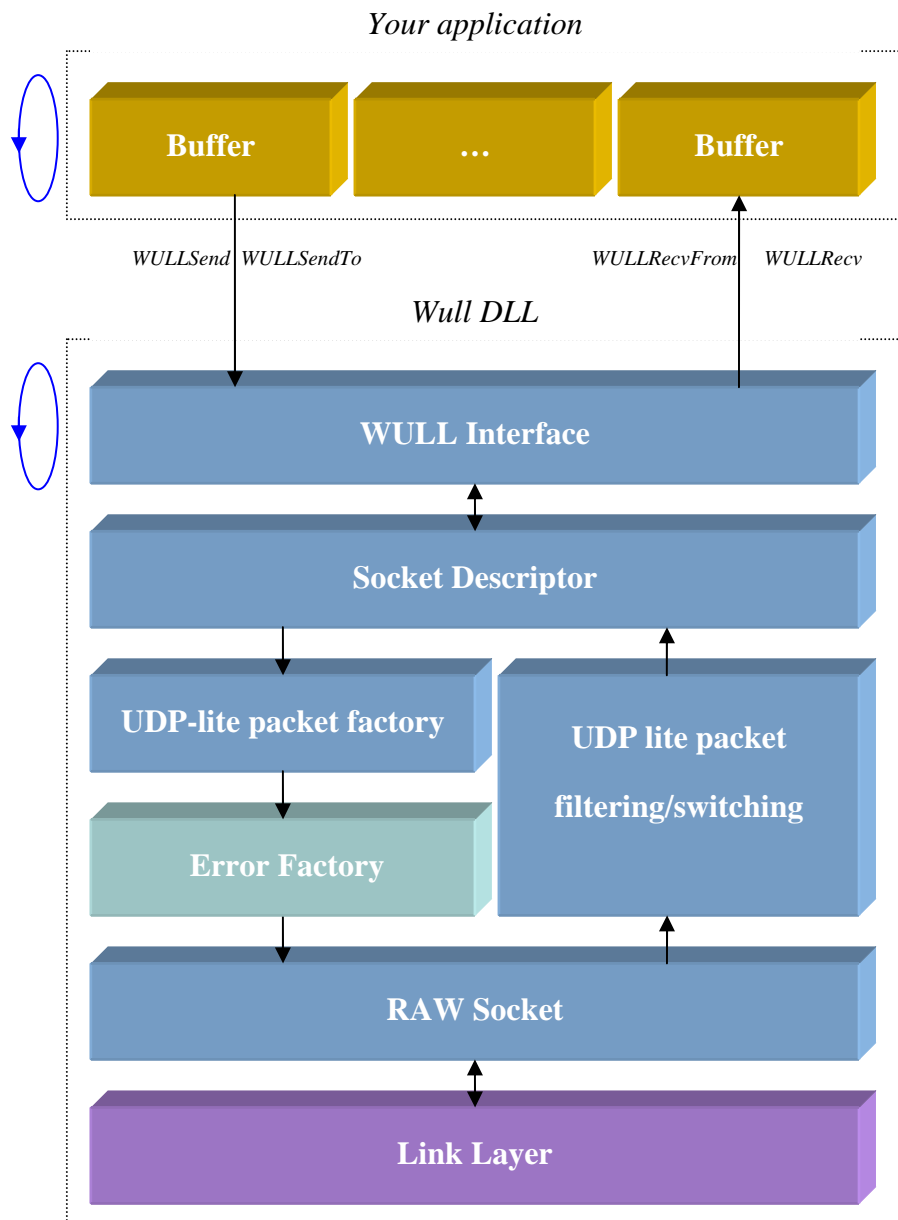


Figure 3: Wull's overall architecture

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

When Wull receives a sending request, data are included in a whole IPv4/UDP-Lite packet: IPv4 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 it an UDP-Lite packet for me, with correct checksum, ...) and if 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 purposes⁷.

As the DLL makes use of socket in raw mode, it only runs on Windows 2000 and Windows XP. Furthermore, applications using Wull must be run with administrator rights.

⁷ Cf. WULLStartup definition (5.19).

5. Wull API

5.1. WULLBind

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

```
int WULLBind(  
    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, **WULLBind** returns zero. Otherwise, it returns **SOCKET_ERROR**, and a specific error code can be retrieved by calling **WULLGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup 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 <code>af_inet</code> case). (See the SO_REUSEADDR socket option under WULLsetsockopt .)
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.

5.2. WULLCleanup

The **WULLCleanup** function terminates use of the Wull.

```
int WULLCleanup(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 **WULLGetLastError**.

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

5.3. WULLCloseEvent

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

```
BOOL WULLCloseEvent(  
    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 **WULLGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup call must occur before using this function.
WSA_INVALID_HANDLE	The <i>hEvent</i> 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**.

5.4. WULLCloseSocket

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

```
int WULLCloseSocket(  
    SOCKET s  
);
```

Parameters

s

[in] Descriptor identifying the socket to close.

Return Values

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

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

5.5. WULLConnect

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

```
int WULLConnect(  
    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, **WULLConnect** returns zero. Otherwise, it returns `SOCKET_ERROR`, and a specific error code can be retrieved by calling **WULLGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup 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.

5.6. WULLCreateEvent

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

```
HANDLE WULLCreateEvent(void);
```

Parameters

This function has no parameters.

Return Values

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

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup 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

5.7. WULLEventSelect

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

```
int WULLEventSelect(  
    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 **WULLGetLastError**.

5.8. WULLGetLastError

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

```
int WULLGetLastError(void);
```

Parameters

This function has no parameters.

Return Values

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

5.9. WULLgetsockname

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

```
int WULLgetsockname(  
    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, **WULLgetsockname** returns zero. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling **WULLGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup 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 WULLbind , or ADDR_ANY is specified in WULLbind but connection has not yet occurs.

5.10. WULLgetsockopt

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

```
int WULLgetsockopt(
    SOCKET s,
    int level,
    int optname,
    void * 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. It could be a pointer to an integer (e.g. the checksum value) or to a char (e.g. the TTL value).

optlen

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

Return Values

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

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

WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The <i>level</i> parameter is unknown or invalid.
WSAENOPROTOOPT	The option is unknown or unsupported by the indicated protocol family.

5.11. WULLioctlsocket

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

```
int WULLioctlsocket (
    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 **WULLioctlsocket** returns zero. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling **WULLGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup 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 WULLEventSelect has been made).

Remarks

The **WULLioctlsocket** 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.

<i>argp</i> value	Nonblocking mode
0	Disabled
nonzero	Enabled

5.12. WULLRecv

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

```
int WULLRecv (
    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 WULL).

Return Values

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

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

WSAEINVAL	The socket has not been bound with WULLBind
WSAENOTCONN	The socket is not connected.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.

Remarks

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

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

For connected or connectionless sockets, the **WULLRecv** 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 (silently) 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 **WULLConnect** function.

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

5.13. WULLRecvFrom

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

```
int WULLRecvFrom (
    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 WULL).

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, **WULLRecvFrom** returns the number of bytes received. If **SOCKET_ERROR** is returned, a specific error code can be retrieved by calling **WULLGetLastError**.

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

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound with WULLBind .
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 <i>buf</i> or <i>from</i> parameters are not part of the user address space, or the <i>fromlen</i> parameter is too small to accommodate the peer address.
WULLSENBADCHECKSUM	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 **WULLRecvFrom** 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 **WULLBind**.

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 **WULLRecvFrom** function blocks and waits for data to arrive set unless the socket is nonblocking. **WULLEventSelect** can be used to determine when more data arrives.

5.14. WULLResetEvent

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

```
BOOL WULLResetEvent(  
  
    WSAEVENT hEvent  
  
);
```

Parameters

hEvent

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

Return Values

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

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup call must occur before using this function.
WSA_INVALID_HANDLE	The <i>hEvent</i> parameter is not a valid event object handle.

Remarks

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

5.15. WULLSend

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

```
int WULLSend (  
  
    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 WULL).

Return Values

If no error occurs, **WULLSend** 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 **WULLGetLastError**.

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

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound with WULLBind .
WSAENOTCONN	The socket is not connected.
WULLENDBADCHECKSUM	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).
WULLENDBUFFERTOOSHORT	The data length to send is too big for the sending buffer .

Remarks

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

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

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

5.16. WULLSendTo

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

```
int WULLsendto (  
    SOCKET s,  
    const char* buf,  
    int len,
```

```

int flags,

const struct sockaddr* to,

int tolen

);

```

Parameters

s

[in] Descriptor identifying a (possibly connected) WULL 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 WULL).

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, **WULLSendTo** 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 **WULLGetLastError**.

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

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEDESTADDRREQ	A destination address is required.
WSAEFAULT	The <i>buf</i> or <i>to</i> parameters are not part of the user address space, or the <i>tolen</i> parameter is too small.
WULLSEENBADCHECKSUM	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).
WULLSEENBUFFERTOOSHORT	The data length to send is too big for the sending buffer .

Remarks

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

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

The **WULLSendTo** 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.

5.17. WULLsetsockopt

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

```
int WULLsetsockopt (  
    SOCKET s,  
    int level,  
    int optname,  
    const void* 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. It could be a pointer to an integer (e.g. the checksum value) or a pointer to a char (e.g. the TTL value)

optlen

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

Return Values

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

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

5.18. WULLSocket

The **WULLSocket** function creates a socket that is bound to a specific transport-service provider. The only transport protocol authorized is `IPPROTO_UDPLITE`.

```
SOCKET WULLSocket (
    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, **WULLSocket** 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 **WULLGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup 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 <code>SOCK_DGRAM</code>)
WSAEPROTONOSUPPORT	The specified protocol is not supported (must be <code>IPPROTO_UDPLITE</code>).

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 **WULLSendTo** and **WULLRecvFrom**. If such a socket is connected to a specific peer, datagrams can be sent to that peer using **WULLSend** and can be received from (only) this peer using **WULLRecv**.

5.19. WULLStartup

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

```
int WULLStartup(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 them⁸.

Return Values

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

An application cannot call **WULLGetLastError** 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
WULLNOKEY	No key has been attributed for this dll. It means that Wull setup has not been properly installed and no license is accorded to wull dll.
WULLINVALIDKEY	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.

⁸ An example function is presented in Annexe B.

WSAEFAULT	The call to WSASStartup has failed.
------------------	-------------------------------------

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

5.20. WULLVirtualSend

The **WULLVirtualSend** function sends virtually data on a connected socket. In fact it does not really send the packet but can be used to simulate sending/receiving.

```
int WULLVirtualSend (  
    SOCKET s,  
    const char* buf,  
    int len,  
    int flags  
);
```

Parameters

s

[in] Descriptor identifying a connected socket.

buf

[in/out] Buffer containing the data to be transmitted; and as output contains the buffer virtually received.

len

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

flags

[in/out] Indicator specifying the way in which the call is made. Should be 0 in input.

In output is zero if the packet is correctly received, and other value if virtually lost..

Return Values

If no error occurs, **WULLVirtualSend** 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 **WULLGetLastError**.

The number of bytes sent corresponds to the number of bytes of the UDPLite payload. **WULLVirtualSend** simulates the sending/receiving of packets in a real channel. Then it uses the error function if one has been defined when calling **WULLStartup**. **The error function is then applied and WULLVirtualSend returns the result when receiving.**

If some error should be detected at receiver (IP or UDP-Lite checksum false), the function change the flags value to a value different from zero.

- If the error is detected in IP, flags is one.
- If the error is detected in IP, flags is two.

This should be interpreted in your developed application, that the packet is lost.

Otherwise, the function changes the buffer in input parameter to the received (eventually corrupted) data.

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound with WULLBind .
WSAENOTCONN	The socket is not connected.
WULLENDBADCHECKSUM	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).
WULLENDBUFFERTOOSHORT	The data length to send is too big for the sending buffer .

Remarks

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

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

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

5.21. WULLVirtualSendTo

The **WULLVirtualSendTo** function sends data to a specific destination. In fact it does not really send the packet but can be used to simulate sending/receiving.

```
int WULLVirtualSendTo (  
    SOCKET s,  
    const char* buf,  
    int len,  
    int flags,  
    const struct sockaddr* to,  
    int tolen  
);
```

Parameters

s

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

buf

[in/out] Buffer containing the data to be transmitted; and as output contains the buffer virtually received.

len

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

flags

[in/out] Indicator specifying the way in which the call is made. Should be 0 in input. In output is zero if the packet is correctly received, and other value if virtually lost..

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, **WULLVirtualSendTo** 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 **WULLGetLastError**.

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

WULLVirtualSendTo simulates the sending/receiving of packets in a real channel. Then it uses the error function if one has been defined when calling **WULLStartup**. **The error function is then applied and WULLVirtualSendTo returns the result when receiving.**

If some error should be detected at receiver (IP or UDP-Lite checksum false), the function change the flags value to a value different from zero.

- If the error is detected in IP, flags is one.
- If the error is detected in IP, flags is two.

This should be interpreted in your developed application, that the packet is lost.

Otherwise, the function changes the buffer in input parameter to the received (eventually corrupted) data.

Error code	Meaning
WSANOTINITIALISED	A successful WULLStartup call must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEDESTADDRREQ	A destination address is required.
WSAEFAULT	The <i>buf</i> or <i>to</i> parameters are not part of the user address space, or the <i>tolen</i> parameter is too small.
WULLSENBADCHECKSUM	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).
WULLENDBUFFERTOOSHORT	The data length to send is too big for the sending buffer .

Remarks

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

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

The **WULLVirtualSendTo** 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. Installation and configuration

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

6.1. Installation

Wull is shipped with a setup utility (“WullSetup.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 Wull partners. Thanks to this key, applications using Wull 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 “WullSetup.exe” and enter your dedicated key:

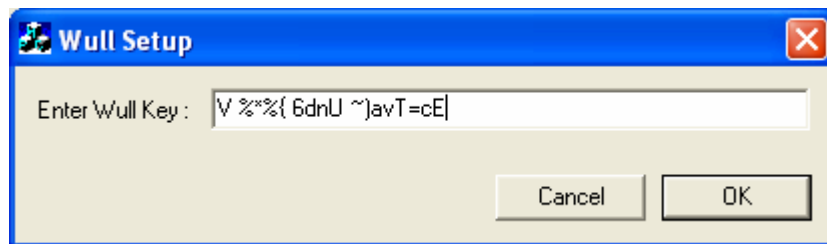


Figure 4: Entering the software key

6.2. Configuration

Once Wull is installed, it can run without further configuration. However, Wull has been designed to make its use as easy as possible whatever your configuration. Two default parameters can be modified:

- the network interface,
- the protocol number dedicated to UDP-Lite.

The two following chapters explain how to do so.

6.2.1. Network interface

Let imagine your PC has several network interface card. By default, Wull uses the first one and the raw socket will be created on that interface⁹. However, if you want Wull to use one of the other cards, you have to specify its rank in the Windows registry.

To do so, you must have administrator rights and execute the “regedit” command. Then set the “HKEY_LOCAL_MACHINES\SOFTWARES\Irisa\Wull\IPCardNumber” to the selected value¹⁰.

6.2.2. Protocol number

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.

⁹ To know which card is the first one, use the “ipconfig” command.

¹⁰ 0 for the first card, 1 for the second and so on.

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.

6.3. Restriction of use

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

- Wull only runs on Windows 2000 and XP
- Only IPv4 in unicast mode is supported for the moment
- Wull must be run with administrator rights
- The maximal number of sockets that Wull can handle is set to 30.

Annexe A Performance: a few figures

Wull runs on Windows 2000 and Windows XP. Numerous tests have been run. Maximum bit rates depend on PC power.

For instance, a 20Mbps bit rate without losses has been achieved with 2 PCs with a crossed link between. The application was a file transfer application. The first PC, the receiver was a 2Ghz PC on Windows 2000 and the second, the transmitter, was a 1,8 Ghz PC on Windows XP.

Annexe B A full sender/receiver example

The following source code is a sender receiver application that transfers a file using UDP-Lite. The bit rate can be set as a parameter. The receiver takes advantage of Windows events to be notified when packets arrive (cf. WaitForMultipleObjects call).

The program is the same for the sender of receiver. However, users have to specify different parameters (Cf. Usage):

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

Moreover, this example uses a bit error generating function, which reads an error mask in file and swaps bit values of the IP/UDP-Lite packets according to this mask¹¹. Masks that are used can be very long and can emulate, for instance, bit errors on a wireless link¹²

```
#include "stdafx.h"

#include <stdio.h>
#include <stdlib.h>
#include <afxmt.h>

#include "wull.h"

char* g_pErrorMask= NULL;
long g_nIndexErrorMask = 0;
long g_nErrorMaskLen;

void InitErrorGenerator(char* pFilename)
{
    FILE* pFile= NULL;

    pFile = fopen(pFilename, "rb");

    if (pFile)
    {
        fseek(pFile, 0, SEEK_END);
        g_nErrorMaskLen = ftell(pFile);

        g_pErrorMask = (char*) malloc(g_nErrorMaskLen);

        fseek(pFile, SEEK_SET, 0);
        fread(g_pErrorMask, sizeof(char), g_nErrorMaskLen , pFile);

        fclose(pFile);
    }
}

int ErrorGenerator(char *pBuffer, int nBufferLen)
{
    for (int i = 0; i < nBufferLen; i++)
    {
        pBuffer[i] = pBuffer[i]^g_pErrorMask[g_nIndexErrorMask++%g_nErrorMaskLen];
    }
    return 0;
}

int main(int argc, char* argv[])
{
    FILE*          pFile = NULL;
```

¹¹ If mask equals 0, keep the previous value, otherwise swap it.

¹² Cf. « Transmission of JPEG 2000 images over a DRM system : error patterns and modelisation od DRM channels », D. Nicholson, C. Lamy-Bergot, X.Naturel, JPEG2000 part 11.

```

SOCKET      sSocketTest = INVALID_SOCKET;
bool        bSender = true;
int         nPort;
int         nRet = 0;
int         nNbOfBytesToSend = 0;
int         nNbMaxOfBytesToSend = 0;
char        sBuffer[1024];
char        ReceiveBuf[1024];
SOCKADDR_IN ReceiverAddr;
SOCKADDR_IN SenderAddr;
int         SenderAddrSize = sizeof(SenderAddr);
long        nlFileLength = 0;
long        nlReceivedLength = 0;
long        nlSentLength = 0;
int         nSleepDuration = 0;
HANDLE      aEventHandle[WSA_MAXIMUM_WAIT_EVENTS];

printf("Test program\n");

if (argc != 6)
{
    printf("USAGE: WullExample <s|r> <filename> <receiver IP address> <port> <rate\n");
kb/s> .\n");
    printf("\t <s|r> : s = sender, r = receiver\n");
    return 0;
}

// initialize WULL
if ((nRet = WULLStartup((int (*)(char* , unsigned int )) ErrorGenerator)) != 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.

    printf("ERROR: WULLStartup failed with error %d\n", nRet);
    return 0;
}

// load error pattern from a file. The erro generator uses that pattern.
InitErrorGenerator(< your file name>

// create a UDP lite socket
sSocketTest = WULLSocket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE);
if (sSocketTest == INVALID_SOCKET)
{
    printf("ERROR: socket failed with error %d\n", WULLGetLastError());
    //WULLCleanup();
}

switch(argv[1][0])
{
    case 's':
        bSender = true;
        break;
    case 'r':
        bSender = false;
        break;
}

nPort = atoi(argv[4]);

if (bSender)
{
    // sender mode

    // Setup a SOCKADDR_IN structure that will identify who we
    // will send datagrams to.

    ReceiverAddr.sin_family = AF_INET;
    ReceiverAddr.sin_port = htons(nPort);
    ReceiverAddr.sin_addr.s_addr = inet_addr(argv[3]);

    // check if the file to transfer exists
    pFile = fopen(argv[2], "rb");

```

```

    if (!pFile)
    {
        printf("File %s not found\n", argv[2]);
        return 0;
    }

    // first send the file length
    if (!fseek(pFile, 0, SEEK_END))
    {
        nlFileLength = ftell(pFile);
        *((long*) sBuffer) = nlFileLength;
        fseek(pFile, SEEK_SET, 0);
    }

    if ((nRet = WULLSendTo(sSocketTest, sBuffer, sizeof(long), 0,
        (SOCKADDR *)&ReceiverAddr,
sizeof(ReceiverAddr))) == SOCKET_ERROR)
    {
        printf("ERROR: sendto failed with error %d\n", WULLGetLastError());
        WULLCloseSocket(sSocketTest);
        // WULLCleanup();
        return 0;
    }

    // compute the number of bytes we have to send per seconds
    nNbOfBytesToSend = atoi(argv[5]) / 8;

    // compute the sleep duration (milli seconds)
    nSleepDuration = 1000*sizeof(sBuffer)/nNbOfBytesToSend;

    // set a timer with which we send data
    LARGE_INTEGER li;
    HANDLE hTimer= CreateWaitableTimer(NULL, FALSE, NULL);

    li.QuadPart=-nSleepDuration*1000*10; // unit: 100 nano seconds
    SetWaitableTimer(hTimer, &li, nSleepDuration, NULL, NULL, false);

    int nTotalSent = 0;

    DWORD dwWait;

    while (nlSentLength < nlFileLength)
    {
        dwWait = WaitForSingleObject(hTimer, INFINITE);

        int nReadBytes = 0;
        nReadBytes = fread(sBuffer, sizeof(char), sizeof(sBuffer), pFile);

        // Send a datagram to the receiver.
        if ((nRet = WULLSendTo(sSocketTest, sBuffer, nReadBytes, 0,
            (SOCKADDR *)&ReceiverAddr,
sizeof(ReceiverAddr))) == SOCKET_ERROR)
        {
            printf("ERROR: sendto failed with error %d\n",
WULLGetLastError());
            WULLCloseSocket(sSocketTest);
            // WULLCleanup();
            return 0;
        }

        nTotalSent+= nRet;

        nlSentLength += nReadBytes;
    }

    // When your application is finished sending datagrams close the socket.
    WULLCloseSocket(sSocketTest);

    // When your application is finished call WULLCleanup.
    //WULLCleanup();

    printf("Tansfer completed, %d bytes sent", nTotalSent);

```

```

    }
    else
    {
        // receiver mode

        // Setup a SOCKADDR_IN structure that will tell bind that we
        // want to receive datagrams from all interfaces using the specified port
        ReceiverAddr.sin_family = AF_INET;
        ReceiverAddr.sin_port = htons(nPort);
        ReceiverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

        // Associate the address information with the socket using bind.
        if (WULBind(sSocketTest, (SOCKADDR *)&ReceiverAddr, sizeof(ReceiverAddr)) ==
SOCKET_ERROR)
        {
            printf("ERROR: bind failed with error %d\n", WULLGetLastError());
            WULLCloseSocket(sSocketTest);
            return 0;
        }

        // create a file to record data
        pFile = fopen(argv[2], "w");
        if (!pFile)
        {
            printf("Reception file not created");
            return 0;
        }

        // first read the file's length
        if ((nRet = WULLRecvFrom(sSocketTest, ReceiveBuf, sizeof(int), 0,
                                (SOCKADDR
*&SenderAddr, &SenderAddrSize)) == SOCKET_ERROR)
        {
            printf("ERROR: recvfrom failed with error %d\n", WULLGetLastError());
            WULLCloseSocket(sSocketTest);
            //WULLCleanup();
            return 0;
        }

        // record the file's size
        nlFileLength = *((long*) ReceiveBuf);

        // create an event
        HANDLE hRecvEvt = WULLCreateEvent();

        // link that event to the socket
        WULLEventSelect(sSocketTest, hRecvEvt, FD_READ);

        aEventHandle[0] = hRecvEvt;
        int nNbOfEvents = 1;

        DWORD dWaitValue;

        while (nlReceivedLength < nlFileLength)
        {
            INFINITE);

            dWaitValue = WaitForMultipleObjects(nNbOfEvents, aEventHandle, FALSE,

            // an event is signaled
            int nIndex = dWaitValue - WAIT_OBJECT_0;
            ASSERT(nIndex >= 0);
            switch(nIndex)
            {
                case 0:
                    //reset the event
                    WULLResetEvent(aEventHandle[0]);
                    // read data
                    if ((nRet = WULLRecvFrom(sSocketTest, ReceiveBuf,
sizeof(ReceiveBuf), 0,
                                (SOCKADDR *)&SenderAddr, &SenderAddrSize)) == SOCKET_ERROR)
                    {

```

```

                                                                    printf("ERROR: recvfrom failed with error %d\n",
WULLGetLastError());
                                                                    closesocket(sSocketTest);
                                                                    //WULLCleanup();
                                                                    return 0;
                                                                    }

                                                                    // write data into the file
                                                                    fwrite(ReceiveBuf, nRet, sizeof(char), pFile);
                                                                    nlReceivedLength += nRet;
                                                                    printf("received bytes : %d\n", nlReceivedLength);
                                                                    break;
                                                                    }
                                                                    }

                                                                    // the application is finished receiving datagrams close the socket
                                                                    WULLCloseSocket(sSocketTest);

                                                                    // When your application is finished call WULLCleanup.
                                                                    //WULLCleanup();

                                                                    printf("Transfer completed: %d bytes received\n", nlReceivedLength);
                                                                    }
                                                                    return 0;
                                                                    }

```

Annexe C Error codes

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

The following is a list of possible error codes returned by the `WULLGetLastError` 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 `WULLBind` 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 `WULLsetsockopt(SO_REUSEADDR)`. Client applications usually need not call `WULLBind` at all - `WULLConnect` will choose an unused port automatically. When `WULLBind` 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 `WULLConnect`.

WSAEADDRNOTAVAIL

(10049)

Cannot assign requested address.

The requested address is not valid in its context. Normally results from an attempt to `WULLBind` to an address that is not valid for the local machine. This can also result from `WULLConnect`, `WULLSendTo` 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 `WULLSocket` call, or if an address of the wrong family is used for a socket, e.g. in `WULLSendTo`.

WSAEDESTADDRREQ

(10039)

Destination address required.

A required address was omitted from an operation on a socket. For example, this error will be returned if `WULLSendTo` 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 **WULLsetsockopt** function).

WSAEISCONN

(10056)

Socket is already connected.

A connect request was made on an already connected socket. Some implementations also return this error if **WULLSendTo** is called on a connected SOCK_DGRAM socket (For SOCK_STREAM sockets, the *to* parameter in **WULLSendTo** 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.

WSAENOPROTOPT

(10042)

Bad protocol option.

An unknown, invalid or unsupported option or level was specified in a **WULLgetsockopt** or **WULLsetsockopt** 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 **WULLSendTo**) no address was supplied. Any other type of operation might also return this error - for example, **WULLsetsockopt** 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 **WULLSocket** call requests a **SOCK_DGRAM** socket, but specifies a stream protocol.

WSAEPROTOTYPE

(10041)

Protocol wrong type for socket.

A protocol was specified in the **WULLSocket** 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 **WULLStartup** or **WULLStartup** 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 **WULLCleanup** has been called too many times.

WULLNOKEY

(10240)

No key has been attributed for this dll. It means that Wull setup has not been properly made and no license is accorded to wull dll.

WULLINVALIDKEY

(10241)

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 .

WULLSENDERBADCHECKSUM

(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).

WULLENDBUFFERTOOSHORT

(10243)

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