## Contents

1 DBL
   1.1 Introduction ......................................................... 1
      1.1.1 Terms and Concepts ............................................. 1
      1.1.2 Example Pseudo-Code .............................. 1
   1.2 Interaction with Sockets ............................................ 3
   1.3 Receive Data Buffering .............................................. 3

2 Module Index .......................................................... 5
   2.1 API Reference ........................................................ 5

3 Namespace Index ......................................................... 7
   3.1 Namespace List ........................................................ 7

4 Data Structure Index .................................................... 9
   4.1 Data Structures ...................................................... 9

5 Module Documentation .................................................... 11
   5.1 API Reference ........................................................ 11
      5.1.1 Detailed Description ........................................... 13
      5.1.2 API Reference ................................................ 13
      5.1.3 Macro Definition Documentation ................................ 13
         5.1.3.1 DBL_VERSION_API ................................. 13
      5.1.4 Enumeration Type Documentation ................................ 13
         5.1.4.1 dbl_filter_mode .................................... 13
         5.1.4.2 dbl_recvmode ....................................... 13
      5.1.5 Function Documentation ......................................... 14
         5.1.5.1 dbl_bind ............................................. 14
         5.1.5.2 dbl_bind_addr ...................................... 14
         5.1.5.3 dbl_close ........................................... 15
         5.1.5.4 dbl_device_enable .................................. 15
         5.1.5.5 dbl_device_get_attrs ............................. 15
         5.1.5.6 dbl_device_handle ................................. 16
         5.1.5.7 dbl_device_set_attrs .............................. 16
         5.1.5.8 dbl_getaddress .................................... 16
         5.1.5.9 dbl_getticks ........................................ 17
         5.1.5.10 dbl_init ............................................. 17
         5.1.5.11 dbl_mcast_block_source ......................... 17
         5.1.5.12 dbl_mcast_join .................................. 18
         5.1.5.13 dbl_mcast_join_source ......................... 18
         5.1.5.14 dbl_mcast_leave ................................ 19
5.1.5.15  `dbl_mcast_leave_source` ................................................................. 19
5.1.5.16  `dbl_mcast_unblock_source` ............................................................ 19
5.1.5.17  `dbl_open` ..................................................................................... 20
5.1.5.18  `dbl_open_if` .................................................................................. 20
5.1.5.19  `dbl_recvfrom` ................................................................................ 21
5.1.5.20  `dbl_send` ...................................................................................... 21
5.1.5.21  `dbl_send_connect` .......................................................................... 22
5.1.5.22  `dbl_send_disconnect` .................................................................... 23
5.1.5.23  `dbl_sendto` ................................................................................... 23
5.1.5.24  `dbl_set_filter_mode` ..................................................................... 23
5.1.5.25  `dbl_shutdown` ............................................................................... 23
5.1.5.26  `dbl_unbind` .................................................................................. 24

5.2  Flags used for `dbl_open()` ................................................................. 25
5.2.1  Detailed Description ............................................................................. 25
5.2.2  Macro Definition Documentation ......................................................... 25
5.2.2.1  `DBL_OPEN_DISABLED` ................................................................. 25
5.2.2.2  `DBL_OPEN_HW_TIMESTAMPING` ...................................................... 25
5.2.2.3  `DBL_OPEN_THREADSAFE` ............................................................... 25

5.3  Flags used for `dbl_bind()` ................................................................. 26
5.3.1  Detailed Description ............................................................................. 26
5.3.2  Macro Definition Documentation ......................................................... 26
5.3.2.1  `DBL_BIND_BROADCAST` ............................................................... 26
5.3.2.2  `DBL_BIND_DUP_TO_KERNEL` .......................................................... 26
5.3.2.3  `DBL_BIND_NO_UNICAST` ................................................................. 26
5.3.2.4  `DBL_BIND_REUSEADDR` ................................................................. 26

5.4  Flags for `dbl_send()` ............................................................................ 27
5.4.1  Detailed Description ............................................................................. 27
5.4.2  Macro Definition Documentation ......................................................... 27
5.4.2.1  `DBL_NONBLOCK` ........................................................................... 27

5.5  Extensions ............................................................................................... 28
5.5.1  Detailed Description ............................................................................. 28
5.5.2  Introduction to extensions .................................................................... 29
5.5.3  Function Documentation .................................................................... 29
5.5.3.1  `dbl_ext_accept` ............................................................................ 29
5.5.3.2  `dbl_ext_channel_type` ................................................................. 29
5.5.3.3  `dbl_ext_getchopt` ....................................................................... 29
5.5.3.4  `dbl_ext_listen` ............................................................................. 30
5.5.3.5  `dbl_ext_poll` ................................................................................. 30
5.5.3.6  `dbl_ext_recv` ................................................................................. 31
5.5.3.7  `dbl_ext_recvmsg` ......................................................................... 31
5.5.3.8  `dbl_ext_send` ................................................................................. 32
5.5.3.9  `dbl_ext_setchopt` ....................................................................... 32

6  Namespace Documentation ...................................................................... 33
6.1  `dbl` Namespace Reference ................................................................. 33
6.1.1  Detailed Description ........................................................................... 33

7  Data Structure Documentation ............................................................ 35
7.1  `dbl_device_attrs` Struct Reference ...................................................... 35
7.1.1  Detailed Description ........................................................................... 35
7.1.2  Field Documentation .......................................................................... 35

Version 3.1.6.52817 ii
7.1.2.1 hw_timestamping ................................................. 35
7.1.2.2 recvq_filter_mode .............................................. 35
7.1.2.3 recvq_size .......................................................... 35

7.2 dbl_recv_info Struct Reference ....................................... 36
  7.2.1 Detailed Description ............................................... 36
  7.2.2 Field Documentation ............................................... 36
    7.2.2.1 chan ............................................................. 36
    7.2.2.2 chan_context .................................................. 36
    7.2.2.3 in_buffer ...................................................... 36
    7.2.2.4 msg_len ......................................................... 36
    7.2.2.5 sin_from ....................................................... 36
    7.2.2.6 sin_to .......................................................... 37
    7.2.2.7 timestamp ..................................................... 37

7.3 dbl_ticks_ Struct Reference .......................................... 37

Index ................................................................................. 38

Version 3.1.6.52817 iii
Chapter 1

DBL

1.1 Introduction

DBL provides a very low-latency interface for sending and receiving UDP datagrams or TCP packets as part of the DBL extensions. The DBL library communicates directly with the firmware on the NIC to send and receive packets, removing the overhead associated with kernel calls and the TCP/UDP stack.

1.1.1 Terms and Concepts

The DBL API uses 3 different entities: "devices", "channels", and "send handles".

A device is the abstraction of a NIC, and there will generally be one device per NIC in a given process. A device is created by calling `dbl_open()`. Several channels can attach to a device.

A channel is roughly the equivalent of a socket opened on a device, with a port number specified. A channel is created by calling `dbl_bind()` on a particular device. When calling `dbl_bind` the type of the channel (e.g TCP or UDP) must be specified.

A send handle is a handle associated with a specific destination that is used to very efficiently send packets to that destination. Send handles are not necessary for sending. A send handle is created by calling `dbl_send_connect()`.

Demultiplexing of incoming data on a device is done by the user code in order to reduce overhead in the library. There is a single call, `dbl_recvfrom()` that will return the next packet available from a given device. A buffer is passed into this function, and any received data will be placed into the buffer upon return. The received packet may be intended for any channel associated with the specified device. A device allows for the mix of UDP or TCP channels.

1.1.2 Example Pseudo-Code

Example use cases:
A device is opened via a call to `dbl_open()`. An interface is specified to `dbl_open` via its first argument which is a struct in_addr. The DBL interface whose IP address matches this address will be opened and a device handle returned.

```
\hyperlink{group__DBL_gab9aed304b284dec7143ff83809a2d6fc}{dbl\_init}();
\hyperlink{group__DBL_gacdc677ef6b2d20f994ad45ca28373768}{dbl\_open}(interface, flags, &dev);
```
The following pseudo-code demonstrates typical multi-port receiver. For each port on which the program wished to receive data, a

dbl_bind() is used to bind a port to a channel. In this example, two different ports are bound, each with a different context value. The context is returned in the dbl_receive_info structure filled in by dbl_recvfrom() and can be used to demultiplex based on the receiving channel.

```
\hyperlink{group__DBL_gab9aed304b284dec7143ff83809a2d6fc}{dbl_init}();
\hyperlink{group__DBL_gacdc677e66bf2d0f99ad45c2a8373768}{dbl_open}(interface, flags, &dev);
\hyperlink{group__DBL_gaaccc222ec7efc1d2ed62f599ce3fd7d}{dbl_bind}(dev, port1, flags, context1, &chan1);
\hyperlink{group__DBL_gaaccc222ec7efc1d2ed62f599ce3fd7d}{dbl_bind}(dev, port2, flags, context2, &chan2);
\textcolor{keywordflow}{while} (!done) {
    \hyperlink{group__DBL_ga7c8fd37a2ca1147707688&c8b6a95bce}{dbl_recvfrom}(dev, mode, buf, maxlen, &info);
    user\_packet\_handler(buf, info.msg\_len, info.chan\_context);
}\}
```

The basic send function is 

```
\hyperlink{group__DBL_gab9aed304b284dec7143ff83809a2d6fc}{dbl_sendto}(). The following pseudo-code demonstrates sending a packet to a destination specified by the address parameter. address is a sockaddr_in as used by socket sendto();
```

```
\hyperlink{group__DBL_ga7c8fd37a2ca1147707688&c8b6a95bce}{dbl_init}();
\hyperlink{group__DBL_gacdc677e66bf2d0f99ad45c2a8373768}{dbl_open}(interface, flags, &dev);
\hyperlink{group__DBL_gaaccc222ec7efc1d2ed62f599ce3fd7d}{dbl_bind}(dev, port1, flags, context1, &chan1);
\hyperlink{group__DBL_gad645185577f2a2fc01278a6d29602733}{dbl_sendto}(chan1, address, buf, buflen, flags);
```

An alternate and slightly faster way to send can be used when you have a known set of destinations to which you are sending. A "send handle" is first created using dbl_send_connect() A send handle is used internally to save precomputed information for sending to that particular destination.

```
\hyperlink{group__DBL_ga7c8fd37a2ca1147707688&c8b6a95bce}{dbl_init}();
\hyperlink{group__DBL_gacdc677e66bf2d0f99ad45c2a8373768}{dbl_open}(interface, flags, &dev);
\hyperlink{group__DBL_gaaccc222ec7efc1d2ed62f599ce3fd7d}{dbl_bind}(dev, flag, context1, &chan1);
\hyperlink{group__DBL_gab1df9a3b4bc9a1fbb2e8a8f166f6c31}{dbl_send_connect}(chan1, address, flags, ttl, &send\_handle);
\hyperlink{group__DBL_gaf169475824a50f2663f5b6f82e084c06}{dbl_send}(send\_handle, buf, buflen, flags);
```

To receive multicast packets, a channel joins the multicast group via 

```
\hyperlink{group__DBL_gab9aed304b284dec7143ff83809a2d6fc}{dbl_init}();
\hyperlink{group__DBL_gacdc677e66bf2d0f99ad45c2a8373768}{dbl_open}(interface, flags, &dev);
\hyperlink{group__DBL_gaaccc222ec7efc1d2ed62f599ce3fd7d}{dbl_bind}(dev, port1, flags, context1, &chan1);
\hyperlink{group__DBL_gadfd63607d172bddd13809042b2b673244}{dbl_mcast_join}(chan1, mcast\_addr, NULL);
\hyperlink{group__DBL_ga7c8fd37a2ca1147707688&c8b6a95bce}{dbl_recvfrom}(dev, mode, buf, maxlen, &info);
user\_packet\_handler(buf, info.msg\_len, info.chan\_context);
```

Each channel may join many multicast groups. The example below will receive packets sent to mcast\_addr1:port1, mcast\_addr2:port1, mcast\_addr1:port2, and mcast\_addr3:port2. The packets sent to port1 will have context = context1 and those to port2 will have context = context2.

```
\hyperlink{group__DBL_ga7c8fd37a2ca1147707688&c8b6a95bce}{dbl_init}();
\hyperlink{group__DBL_gacdc677e66bf2d0f99ad45c2a8373768}{dbl_open}(interface, flags, &dev);
\hyperlink{group__DBL_gaaccc222ec7efc1d2ed62f599ce3fd7d}{dbl_bind}(dev, port1, flags, context1, &chan1);
\hyperlink{group__DBL_gaaccc222ec7efc1d2ed62f599ce3fd7d}{dbl_bind}(dev, port2, flags, context2, &chan2);
\hyperlink{group__DBL_gadfd63607d172bddd13809042b2b673244}{dbl_mcast_join}(chan1, mcast\_addr1, NULL);
\hyperlink{group__DBL_gadfd63607d172bddd13809042b2b673244}{dbl_mcast_join}(chan1, mcast\_addr2, NULL);
\hyperlink{group__DBL_gadfd63607d172bddd13809042b2b673244}{dbl_mcast_join}(chan2, mcast\_addr1, NULL);
\hyperlink{group__DBL_gadfd63607d172bddd13809042b2b673244}{dbl_mcast_join}(chan2, mcast\_addr2, NULL);
\hyperlink{group__DBL_ga7c8fd37a2ca1147707688&c8b6a95bce}{dbl_recvfrom}(dev, mode, buf, maxlen, &info);
user\_packet\_handler(buf, info.msg\_len, info.chan\_context);
```
1.2 Interaction with Sockets

Since DBL packets move straight from the NIC to the user-level library, there is generally no opportunity for these packets to be shared with other processes using the socket interface. Thus, under default conditions, if a process using the DBL API and one using the socket API both open and bind to the same address (using appropriate REUSEADDR-style flags), only the DBL process will actually receive the packets. This is because the packets are never delivered to the kernel and the DBL process has no way to know that another process is listening for the packets.

In order to allow sockets-based processes to receive packets that are being received by DBL processes, the DBL process must not only specify the DBL_BIND_REUSE_ADDR flag to `dbl_bind()`, it must also specify the DBL_BIND_DUP_TO_KERNEL flag which will cause the firmware on the NIC to duplicate each packet to the kernel UDP stack for possible delivery to any sockets-based processes wishing to receive them. Note that this duplication will happen for every packet delivered to the socket address (IP and port number) specified in the call to `dbl_bind` with the DUP_TO_KERNEL flag, regardless of whether there is a socket application bound to the address or not.

Specifying DBL_BIND_DUP_TO_KERNEL will add 1.8 us or less to each packet whose destination is the address specified in the `dbl_bind()` call.

1.3 Receive Data Buffering

There are two different places that packets are buffered in DBL. The first level of buffering is a 48k buffer onboard on the NIC. This buffer is used directly by the hardware on the NIC and is serviced independently of activity on the host.

The second level of buffering is in host memory, and is on a per-device basis, since `dbl_recvfrom` reads from a `dbl_device_t`. This is a circular buffer which defaults to 128Mb on Linux (the size of the buffer can be changed, see `recvq_size` in `dbl_device_attrs` and `dbl_device_set_attrs`). The NIC asynchronously moves data into this buffer, and the only involvement required from the host is to drain data from this buffer.

On the host buffer, each packet has its length rounded up to a multiple of 64 bytes. Since ethernet packets are a minimum of 64 bytes on lengths and there is bookkeeping data included with the packet, each packet occupies a minimum of 128 bytes of buffer space. This translates to a worst-case capacity of one million packets, or 64 megabytes of data, or roughly 64 milliseconds worth of minimum-sized packets.

There are two different counters that indicate when packets are dropped due to lack of buffering. The first counter, "Net overflow drop" indicates that packets are arriving faster than the NIC can process them. The second counter, "Receive Queue full," indicates that the user application is not draining packets from the host queue quickly enough.
Chapter 2

Module Index

2.1 API Reference

Here is a list of all modules:

API Reference ................................................................. 11
  Flags used for dbl_open() .............................................. 25
  Flags used for dbl_bind() .............................................. 26
  Flags for dbl_send() ................................................... 27
Extensions ................................................................. 28
Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

- dbl

33
Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

- `dbl_device_attrs` ................................................................. 35
- `dbl_recv_info` ................................................................. 36
  Information about the packet received
- `dbl_ticks_` ................................................................. 37
Chapter 5

Module Documentation

5.1 API Reference

API Reference for DBL.

Data Structures

- struct dbl_device_attrs
- struct dbl_recv_info
  
  *Information about the packet received.*

Modules

- Flags used for dbl_open()
- Flags used for dbl_bind()
- Flags for dbl_send().

Macros

- #define DBL_VERSION_API 0x0004

Enumerations

- enum dbl_filter_mode { DBL_RECV_FILTER_NORMAL = 0, DBL_RECV_FILTER_ALLMULTI = 1, DBL_RECV_FILTER_RAW = 2 }
- enum dbl_recvmode {
  DBL_RECV_DEFAULT = 0, DBL_RECV_NONBLOCK = 1, DBL_RECV_BLOCK = 2, DBL_RECV_PEEK = 3, DBL_RECV_PEEK_MSG = 4 }
Functions

- `dbl_init (uint16_t api_version)`
  Initializes the dbl library.

- `dbl_open (const struct in_addr *interface_addr, int flags, dbl_device_t *dev_out)`
  Creates an instance of a `dbl_device`.

- `dbl_open_if (const char *ifname, int flags, dbl_device_t *dev_out)`
  Creates an instance of a `dbl_device`.

- `dbl_device_get_attrs (dbl_device_t dev, struct dbl_device_attrs *attr)`

- `dbl_device_set_attrs (dbl_device_t dev, const struct dbl_device_attrs *attr)`

- `dbl_device_enable (dbl_device_t dev)`

- `dbl_set_filter_mode (dbl_device_t dev, enum dbl_filter_mode mode)`

- `dbl_device_handle (dbl_device_t dev)`
  Returns a descriptor for use with `poll()` or `select()`.

- `dbl_close (dbl_device_t dev)`
  Close a `dbl` device.

- `dbl_bind (dbl_device_t dev, int flags, int port, void *context, dbl_channel_t *handle_out)`
  Create a channel on `dbl` device.

- `dbl_bind_addr (dbl_device_t dev, const struct in_addr *ipaddr, int flags, int port, void *context, dbl_channel_t *handle_out)`
  Creates a channel, using specified ip address.

- `dbl_unbind (dbl_channel_t handle)`
  Destroys a channel.

- `dbl_getaddress (dbl_channel_t ch, struct sockaddr_in *sin)`
  Returns the address to which a channel is bound.

- `dbl_getticks (dbl_device_t dev, dbl_ticks_t *ticks)`
  Returns the current NIC time. It reports both values, NIC ticks and time in usec since epoch.

- `dbl_mcast_join (dbl_channel_t ch, const struct in_addr *mcast_addr, void *unused)`
  Join a multicast group.

- `dbl_mcast_leave (dbl_channel_t ch, const struct in_addr *mcast_addr)`
  Leave a multicast group.

- `dbl_mcast_join_source (dbl_channel_t ch, const struct in_addr *mcast_addr, const struct in_addr *src)`
  Join a multicast group on a given source address.

- `dbl_mcast_leave_source (dbl_channel_t ch, const struct in_addr *mcast_addr, const struct in_addr *src)`
  Leave a multicast group.

- `dbl_mcast_block_source (dbl_channel_t ch, const struct in_addr *join_addr, const struct in_addr *block_addr)`
  block sender.

- `dbl_mcast_unblock_source (dbl_channel_t ch, const struct in_addr *join_addr, const struct in_addr *block_addr)`
  unblock sender.

- `dbl_shutdown (dbl_device_t dev, int how)`
  Unblock `dbl_recvfrom/dbl_ext_recvmsg`.

- `dbl_recvfrom (dbl_device_t dev, enum dbl_recvmode mode, void *buf, size_t len, struct dbl_recv_info *info)`
  Receive data.
• **dbl_send_connect** (dbl_channel_t chan, const struct sockaddr_in *dest_sin, int flags, int ttl, dbl_send_t *hsend)

  Create a send handle for faster sending.

• **dbl_send** (dbl_send_t sendh, const void *buf, size_t len, int flags)

  Send a packet using a send handle.

• **dbl_send_disconnect** (dbl_send_t hsend)

  Release a send handle.

• **dbl_sendto** (dbl_channel_t ch, const struct sockaddr_in *sin, const void *buf, size_t len, int flags)

  Send a packet.

### 5.1.1 Detailed Description

API Reference for DBL.

### 5.1.2 API Reference

### 5.1.3 Macro Definition Documentation

#### 5.1.3.1 #define DBL_VERSION_API 0x0004

DBL API version number (16 bits) Least significant byte increases for minor backwards compatible changes in the API. Most significant byte increases for incompatible changes in the API

- 0x0002: Added timestamp to dbl_recv_info
- 0x0003: Added buflen to dbl_recv_info

### 5.1.4 Enumeration Type Documentation

#### 5.1.4.1enum dbl_filter_mode

Filtering modes (advanced functionality).

Remarks

Selecting anything but the NORMAL filter causes all other DBL devices to be deprived of data. The ALLMULTI and RAW modes cause all matching data from the underlying port to be delivered to the one endpoint. The OS-setting of dup to kernel is honored with all filtering modes, albeit with the same performance constraints.

#### 5.1.4.2 enum dbl_recvmode

Specifies behavior of the dbl_recvfrom call

Enumerator

- **DBL_RECV_DEFAULT**  Busy poll forever until a packet is received.
- **DBL_RECV_NONBLOCK**  Return a packet if available, else return EAGAIN.
- **DBL_RECV_BLOCK**  Block until a packet is available, sleep until interrupt if necessary.
**DBL_RECV_PEEK**  Check for a packet one time, return info, or EAGAIN if no packet.

**DBL_RECV_PEEK_MSG**  Peek but also copy data, return info, or EAGAIN if no packet. Unsupported in the DBL TCP extensions

### 5.1.5  Function Documentation

#### 5.1.5.1  dbl_bind ( dbl_device_t dev, int flags, int port, void * context, dbl_channel_t * handle_out )

Create a channel on dbl device.

Creates a channel on a specified device through which UDP datagrams or TCP streams (if using the DBL TCP extensions), may be sent and received. Any packets sent through this channel will have "port" as their source port and packets arriving on the interface addressed to "port" will be received on this channel. By default, only unicast packets, not broadcast or multicast, will be received on the channel.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dev</code></td>
<td>A DBL device handle returned by a call to <code>dbl_open()</code>.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>See Flags used for <code>dbl_bind()</code>.</td>
</tr>
<tr>
<td><code>port</code></td>
<td>The port to send/receive on.</td>
</tr>
<tr>
<td><code>context</code></td>
<td>The value of context is returned on future receives on this channel.</td>
</tr>
<tr>
<td><code>handle_out</code></td>
<td>The handle to the created channel.</td>
</tr>
</tbody>
</table>

**Return values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0</code></td>
<td>Success</td>
</tr>
<tr>
<td><code>EINVAL</code></td>
<td>Error in arguments</td>
</tr>
<tr>
<td><code>EEXIST</code></td>
<td>Port already in use</td>
</tr>
<tr>
<td><code>?</code></td>
<td>Other values indicate various OS failures in the bind process</td>
</tr>
</tbody>
</table>

If `dbl_bind()` on UDP is called multiple times on the same port on a single device, unicast packets will only be delivered to the oldest channel currently bound to the port. `dbl_bind()` on TCP can only be used exclusively per port.

**Remarks**

This function can be used in the context of DBL TCP API, with some restriction. The `DBL_BIND_DUP_TO_KERNEL` and `DBL_BIND_NO_UNICAST` options are not supported.

#### 5.1.5.2  dbl_bind_addr ( dbl_device_t dev, const struct in_addr * ipaddr, int flags, int port, void * context, dbl_channel_t * handle_out )

Creates a channel, using specified ip address.

Creates a channel on a specified device, just like `dbl_bind`, except that it associates the channel with the specified address instead of the one specified in the `dbl_open` call.

The address used must correspond to an OS-level interface that maps to the same underlying Ethernet port as the interface specified in `dbl_open`. For example, this can be a VLAN interface.

**Parameters**
A DBL device handle returned by a call to `dbl_open()`.

Specifies the IP address of the interface with which the channel created will be associated. This must be on the same underlying interface as the one used in the `dbl_open` call.

See Flags used for `dbl_bind()`.

The port to send/receive on.

The value of context is returned on future receives on this channel.

The handle to the created channel.

| Return values |
|---|---|
| 0 | Success |
| EINVAL | Error in arguments. Specifying an address that is not on the same underlying interface as that specified with `dbl_open` will return EINVAL. |
| EEXIST | Port already in use |
| ? | Other values indicate various OS failures in the bind process |

**Remarks**

DBL TCP supported

### 5.1.5.3 `dbl_close ( dbl_device_t dev )`

Close a dbl device.

Terminate usage of a device returned by `dbl_open()` and free all resources associated with it.

#### Parameters

| dev | The device handle returned from `dbl_open()`.

#### Return values

| 0 | Success |

### 5.1.5.4 `dbl_device_enable ( dbl_device_t dev )`

Function to enable a device if opened with DBL_OPEN_DISABLED

#### Remarks

If this call fails, the user is still responsible for calling `dbl_close()` on the underlying device to free resources

### 5.1.5.5 `dbl_device_get attrs ( dbl_device_t dev, struct dbl_device_attrs *attr )`

Function to retrieve device attributes.

#### Parameters

| dev | The device handle returned from `dbl_open()` |

| attr | Device attributes will be copied out. |
Remarks

Can be used before and after calls that open and enable DBL devices.

5.1.5.6 dbl_device_handle (dbl_device_t dev)

Returns a descriptor for use with poll() or select(). Returns an OS-specific file descriptor which can be passed to poll() or select() to block on receive data available. For UNIX systems, this is a file descriptor, on Windows it is a HANDLE.

Parameters

- *dev* The DBL device whose OS handle is needed.

Returns

OS-specific handle for device

5.1.5.7 dbl_device_set_attrs (dbl_device_t dev, const struct dbl_device_attrs *attr)

Function to set device attributes before a device is enabled.

Parameters

- *dev* The device handle returned from dbl_open() with flag DBL_OPEN_DISABLED.
- *attr* Device attributes that will be set on the device.

Remarks

Can’t be called without having the contents of attr previously filled out by a call to dbl_device_get_attrs. The implementation can change the size of requests to accommodate internal alignment and sizing requirements. If these sizes are changed, the new sizes are reflected during a subsequent call to dbl_device_get_attrs.

5.1.5.8 dbl_getaddress (dbl_channel_t ch, struct sockaddr_in *sin)

Returns the address to which a channel is bound.

Parameters

- *ch* Specifies the channel whose bind information is required.
- *sin* sockaddr_in to which the address will be copied out.

Return values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Bad channel specified</td>
</tr>
</tbody>
</table>
Remarks

DBL TCP supported

5.1.5.9  dbl_getticks ( dbl_device_t dev, dbl_ticks_t *ticks )

Returns the current NIC time. It reports both values, NIC ticks and time in usec since epoch.

Parameters

<table>
<thead>
<tr>
<th>dev</th>
<th>Specifies the dev channel from dbl_open</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticks</td>
<td>Specifies the dbl_ticks_t structure holding the timing information</td>
</tr>
</tbody>
</table>

Return values

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Bad dev specified</td>
</tr>
</tbody>
</table>

Remarks

DBL TCP supported

Under TA, a ioctl/WSAIoctl socket call can use cmd SIO_GETNICTIME

5.1.5.10  dbl_init ( uint16_t api_version )

Initializes the dbl library.

Parameters

| api_version | Must always be DBL_VERSION_API. This is used to ensure compatibility between the application binary and the DBL library. |

Return values

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Bad/incompatible version passed</td>
</tr>
</tbody>
</table>

Remarks

dbl_init() must be called once at the start of any application that uses DBL.

5.1.5.11  dbl_mcast_block_source ( dbl_channel_t ch, const struct in_addr *join_addr, const struct in_addr *block_addr )

block sender.

Indicates that the specified channel wishes to stop receiving packets from a given source and therefore block that sender

Prerequisites: prior call to dbl_mcast_join on same multicast address.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Handle for the channel to leave the specified multicast group.</td>
</tr>
<tr>
<td>join_addr</td>
<td>Address of the multicast group to join.</td>
</tr>
<tr>
<td>block_addr</td>
<td>Address to block. The multicast packets will not be received from the blocked source.</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Argument error, such as address not multicast group.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>Not currently joined to group “address”</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Internal resources temporarily unavailable, try again.</td>
</tr>
<tr>
<td>?</td>
<td>Other non-zero codes indicate various OS failures in the leave process.</td>
</tr>
</tbody>
</table>

5.1.5.12 `dbl_mcast_join ( dbl_channel_t ch, const struct in_addr * mcast_addr, void * unused )`

Join a multicast group.

Indicates that the specified channel wishes to receive packets addressed to the multicast address specified.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Handle for the channel to add to the specified multicast group.</td>
</tr>
<tr>
<td>mcast_addr</td>
<td>Address of the multicast group to join.</td>
</tr>
<tr>
<td>unused</td>
<td>A temporary unused pointer to maintain binary compatibility.</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Argument error, such as address is not a multicast group.</td>
</tr>
<tr>
<td>?</td>
<td>Other values indicate various OS specific failures in the join process.</td>
</tr>
</tbody>
</table>

5.1.5.13 `dbl_mcast_join_source ( dbl_channel_t ch, const struct in_addr * mcast_addr, const struct in_addr * src )`

Join a multicast group on a given source address.

Indicates that the specified channel wishes to receive packets addressed to the multicast address specified from a specific source. For multiple sources, call this function again with the desired sources to receive from.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Handle for the channel to add to the specified multicast group.</td>
</tr>
<tr>
<td>mcast_addr</td>
<td>Address of the multicast group to join.</td>
</tr>
<tr>
<td>src</td>
<td>Address of source to receive multicast from</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Argument error, such as address is not a multicast group.</td>
</tr>
<tr>
<td>?</td>
<td>Other values indicate various OS specific failures in the join process.</td>
</tr>
</tbody>
</table>
5.1.5.14  dbl_mcast_leave (dbl_channel_t ch, const struct in_addr *mcast_addr)

Leave a multicast group.
Indicates that the specified channel wishes to stop receiving packets addressed to the multicast address specified.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Handle for the channel to leave the specified multicast group.</td>
</tr>
<tr>
<td>mcast_addr</td>
<td>Address of the multicast group to leave.</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Argument error, such as address not multicast group.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>Not currently joined to group “address”</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Internal resources temporarily unavailable, try again.</td>
</tr>
<tr>
<td>?</td>
<td>Other non-zero codes indicate various OS failures in the leave process</td>
</tr>
</tbody>
</table>

5.1.5.15  dbl_mcast_leave_source (dbl_channel_t ch, const struct in_addr *mcast_addr, const struct in_addr *src)

Leave a multicast group.
Indicates that the specified channel wishes to stop receiving packets addressed to the multicast address specified.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Handle for the channel to leave the specified multicast group.</td>
</tr>
<tr>
<td>mcast_addr</td>
<td>Address of the multicast group to leave.</td>
</tr>
<tr>
<td>src</td>
<td>Address of the source to drop</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Argument error, such as address not multicast group.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>Not currently joined to group “address”</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Internal resources temporarily unavailable, try again.</td>
</tr>
<tr>
<td>?</td>
<td>Other non-zero codes indicate various OS failures in the leave process</td>
</tr>
</tbody>
</table>

5.1.5.16  dbl_mcast_unblock_source (dbl_channel_t ch, const struct in_addr *join_addr, const struct in_addr *block_addr)

unblock sender.
Indicates that the specified channel wishes to unblock a sender. Receiving packets will commence from the unblocked sender Prerequisites: prior call to dbl_mcast_join on same multicast address. Prior call to dbl_mcast_block_source.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Handle for the channel to leave the specified multicast group.</td>
</tr>
<tr>
<td>join_addr</td>
<td>Address of the multicast group to join.</td>
</tr>
<tr>
<td>block_addr</td>
<td>Address to unblock. The multicast packets will again be received from the unblocked source</td>
</tr>
</tbody>
</table>
Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Argument error, such as address not multicast group.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>Not currently joined to group &quot;address&quot;</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Internal resources temporarily unavailable, try again.</td>
</tr>
<tr>
<td>?</td>
<td>Other non-zero codes indicate various OS failures in the leave process</td>
</tr>
</tbody>
</table>

5.1.5.17  `dbl_open ( const struct in_addr *interface_addr, int flags, dbl_device_t *dev_out )`

Creates an instance of a `dbl_device`.

Creates an instance of a `dbl` device which can be used to subsequently open channels via `dbl_bind()`.

Parameters

- `interface_addr`: Specifies the IP address of the interface with which channels created using `dbl_bind()` will be associated.
- `flags`: A bitmask of flags to alter open behavior. See Flags used for `dbl_open()`.
- `dev_out`: On successful return, this is where the handle for the newly opened device will be placed.

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Bad usage. Includes <code>dbl_init</code> not called first and bad <code>interface_addr</code>.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>No matching IP address found on <code>DBL-enabled NIC</code></td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Internal resources temporarily unavailable, try again.</td>
</tr>
</tbody>
</table>

Remarks

Unlike traditional sockets, a `DBL` channel cannot be associated with multiple network interfaces.

Using the TCP extensions, `dbl_open` opens an endpoint on which several channels of type `UDP` and `TCP` can be demultiplexed.

5.1.5.18  `dbl_open_if ( const char *ifname, int flags, dbl_device_t *dev_out )`

Creates an instance of a `dbl_device`.

Like `dbl_open()` except it takes an interface name instead of an `ip address`.

Parameters

- `ifname`: Specifies the name of the interface with which channels created using `dbl_bind()` will be associated.
- `flags`: A bitmask of flags to alter open behavior. See Flags used for `dbl_open()`.
- `dev_out`: On successful return, this is where the handle for the newly opened device will be placed.

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Bad usage. Includes <code>dbl_init</code> not called first and bad <code>interface_addr</code>.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Internal resources temporarily unavailable, try again.</td>
</tr>
</tbody>
</table>
Remarks

Unlike traditional sockets, a DBL channel cannot be associated with multiple network interfaces.

5.1.5.19 dbi_recvfrom ( dbl_device_t dev, enum dbl_recvmode mode, void * buf, size_t len, struct dbl_recv_info * info )

Receive data.
Used to check for and read data from the channels associated with a particular dbl_device.

Parameters

<table>
<thead>
<tr>
<th>dev</th>
<th>The underlying device via dbl_open</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>See dbl_recvmode</td>
</tr>
<tr>
<td>buf</td>
<td>Buffer in which to place received data.</td>
</tr>
<tr>
<td>len</td>
<td>Maximum number of bytes to write into buf.</td>
</tr>
<tr>
<td>info</td>
<td>See dbl_recv_info.</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Returned if using mode DBL_RECV_NONBLOCK or DBL_RECV_PEEK when no packet is available.</td>
</tr>
<tr>
<td>EINTR</td>
<td>in case dbl_shutdown() was called</td>
</tr>
<tr>
<td>?</td>
<td>Other codes indicate various OS failures.</td>
</tr>
</tbody>
</table>

Remarks

dbl_recvfrom() will, by default, busy-poll checking for data available on the device. This consumes 100% of the CPU available to this single thread, but also guarantees the lowest possible latency for packet delivery. A blocking mode of operation may be specified through the recv_mode parameter, reducing CPU load at the expense of a few microseconds of message latency.

DBL TCP supported. Receiving a return value of 0 with a msg_len of 0 means the channel is disconnected.
On endpoints with mixed channels e.g DBL and DBL extension (TCP) channels the DBL channels are prioritized to avoid packet drops.

5.1.5.20 dbi_send ( dbl_send_t sendh, const void * buf, size_t len, int flags )

Send a packet using a send handle.

Sends a packet to the address associated with the specified send handle. The send_handle must have been previously created by a call to dbl_send_connect(). If internal resources are unavailable to execute the send immediately, the send call will block until resources are available to proceed.

Parameters

<table>
<thead>
<tr>
<th>sendh</th>
<th>Send handle specifying destination for packe.</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>The data to send.</td>
</tr>
<tr>
<td>len</td>
<td>The number of bytes to send.</td>
</tr>
<tr>
<td>flags</td>
<td>See Flags for dbl_send().</td>
</tr>
</tbody>
</table>
Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>DBL_NONBLOCK specified and no resources available.</td>
</tr>
<tr>
<td>?</td>
<td>Other codes indicate various OS failures in the send process.</td>
</tr>
</tbody>
</table>

Remarks

DBL TCP supported with no special flags. The function will block until all data has been transferred. For advanced handling use dbl_ext_send for TCP channels

5.1.5.21  
`dbl_send_connect ( dbl_channel_t chan, const struct sockaddr_in *dest_sin, int flags, int ttl, dbl_send_t *hsend )`

Create a send_handle for faster sending.

Used to create a send handle for fast sending to a remote destination.

Parameters

- **chan**: The channel to be associated with this send handle.
- **dest_sin**: Destination address of packets sent using this handle.
- **flags**: Bitmask of flags to modify default send_connect operation. Currently no flags are supported.
- **ttl**: The value to put in the TTL field of the IP header.
- **hsend**: The send_handle to be used in future calls to `dbl_send()` is returned here.

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Errors in arguments</td>
</tr>
<tr>
<td>?</td>
<td>Other codes indicate various OS failures in the send process.</td>
</tr>
</tbody>
</table>

Remarks

The returned send handle is a reference to a set of precomputed data that is needed to send a packet to a particular destination. This precomputed data is saved and cached by DBL as a matter of course through the `dbl_sendto()` function, but holding a send_handle avoids the need for a hash lookup to find the necessary information. This can take 100-200 ns off the time required to do a send.

Since `dbl_send_connect` will re-use a cached send handle to the same destination, the ttl parameter, if non-zero, will overwrite the ttl value in the cached sendhandle. This means that any future `dbl_sendto` operations to the same destination will use the new ttl value. This also means that if there is a need to use `dbl_sendto` with a different ttl than the default, it is possible to use a call to `dbl_send_connect` to change the ttl.

DBL TCP supported. One can use the dbl semantics (reuse the exact same call, besides the ttl value) to retrieve a send handle, or one can specify a NULL value for dest_sin to retrieve a new send handle which could be clearer in the code than keeping the dest_sin value.

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EISCONN</td>
<td>Channel already connected</td>
</tr>
</tbody>
</table>
5.1.5.22  **dbl_send_disconnect ( dbl_send_t hsend )**

Release a send handle.
Release the resources associated with a send handle.

**Parameters**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hsend</strong></td>
<td>The send handle.</td>
</tr>
</tbody>
</table>

**Return values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
</tbody>
</table>

**Remarks**

DBL TCP supported - in this case the connected peer will receive an EOF which will show up with a msg of len 0. The local channel is re-transitioned into the unconnected state and can be used again in dbl_send_connect.

5.1.5.23  **dbl_sendto ( dbl_channel_t ch, const struct sockaddr * sin, const void * buf, size_t len, int flags )**

Send a packet.
Send a packet to the address specified.

**Parameters**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ch</strong></td>
<td>Handle for the channel to send over.</td>
</tr>
<tr>
<td><strong>sin</strong></td>
<td>The destination address</td>
</tr>
<tr>
<td><strong>buf</strong></td>
<td>The data to send.</td>
</tr>
<tr>
<td><strong>len</strong></td>
<td>The lenght of the data to send.</td>
</tr>
<tr>
<td><strong>flags</strong></td>
<td>See Flags for dbl_send().</td>
</tr>
</tbody>
</table>

**Return values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>DBL_NONBLOCK specified and no resources available.</td>
</tr>
<tr>
<td>?</td>
<td>Other codes indicate various OS failures in the send process.</td>
</tr>
</tbody>
</table>

5.1.5.24  **dbl_set_filter_mode ( dbl_device_t dep, enum dbl_filter_mode mode )**

Function to control per-port DBL filtering modes (advanced functionality).

5.1.5.25  **dbl_shutdown ( dbl_device_t dev, int how )**

Unblock dbl_recvfrom/dbl_ext_recvmsg.
Used to unblock a blocking dbl_recvfrom/dbl_ext_recvmsg.

**Parameters**
### Remarks

**DBL UDP and TCP**

### 5.1.5.26 `dbl_unbind (dbl_channel_t handle)`

Destroys a channel.
Destroys a channel and releases all the resources associated with it.

#### Parameters

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>The handle of the channel to unbind.</td>
</tr>
</tbody>
</table>

#### Return values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
</tbody>
</table>

#### Remarks

**DBL TCP supported**
5.2 Flags used for dbl_open()

Macros

- #define DBL_OPEN_THREADSAFE 0x1
- #define DBL_OPEN_DISABLED 0x2
- #define DBL_OPEN_HW_TIMESTAMPING 0x4

5.2.1 Detailed Description

5.2.2 Macro Definition Documentation

5.2.2.1 #define DBL_OPEN_DISABLED 0x2

A device can be opened but separately enabled through dbl_device_enable. This allows users to change the size of buffers or other properties before it is enabled and ready to receive packets. By setting this flag, users are required to separately call dbl_device_enable after, perhaps, having changed device attributes using dbl_device_get_attrs followed by dbl_device_set_attrs.

5.2.2.2 #define DBL_OPEN_HW_TIMESTAMPING 0x4

Request that incoming packets provide a hardware timestamp to indicate when the packet was received by the NIC. The timestamp provided is a conversion from raw NIC nanoseconds to host nanoseconds as would be returned by gettimeofday(). Unless HW timestamping is requested, packets will return a timestamp of 0.

Alternatively, users can enable/disable the HW timestamping once the device is opened by using dbl_device_get_attrs followed by dbl_device_set_attrs.

5.2.2.3 #define DBL_OPEN_THREADSAFE 0x1

Used to indicate that multiple threads will be using this device, and that locking should be used internally to serialize access. Thread safety is off by default in order to improve performance for the single-threaded case.
5.3 Flags used for dbl_bind()

Macros

- #define DBL_BIND_REUSEADDR 0x02
- #define DBL_BIND_DUP_TO_KERNEL 0x04
- #define DBL_BIND_NO_UNICAST 0x08
- #define DBL_BIND_BROADCAST 0x10

5.3.1 Detailed Description

5.3.2 Macro Definition Documentation

5.3.2.1 #define DBL_BIND_BROADCAST 0x10

Allows this channel to receive broadcast packets.

5.3.2.2 #define DBL_BIND_DUP_TO_KERNEL 0x04

Allows packets to be shared with sockets. (See Interaction with Sockets)

5.3.2.3 #define DBL_BIND_NO_UNICAST 0x08

Instructs this channel not to receive packets addressed to the unicast address.

5.3.2.4 #define DBL_BIND_REUSEADDR 0x02

Allows other dbl_bind() and bind() calls on the same port to succeed.
5.4 Flags for dbl_send().

Macros

• #define DBL_NONBLOCK 0x4

5.4.1 Detailed Description

5.4.2 Macro Definition Documentation

5.4.2.1 #define DBL_NONBLOCK 0x4

Return EAGAIN if send request would block for resources
5.5 Extensions

API extensions for DBL.

Macros

- `#define DBL_FUNC(type) type`
- `#define DBL_VAR(type) type`
- `#define DBL_PROTO_IS_MTCP(flags) ((flags & (1 << 7)) == 0)`
- `#define DBL_TYPE_IS_TCP(flags) ((flags & (1 << 8)) == 0)`
- `#define DBL_INITFLAGS(type, proto) (type << 8 | proto << 7)`
- `#define DBL_TCP 1`
- `#define DBL_UDP 0`
- `#define DBL_BSD 1 /* use the BSD stack */`
- `#define DBL_MYRI 0 /* use the DBL_API for UDP */`
- `#define DBL_CHANNEL_FLAGS(type, proto) DBL_INITFLAGS(type, proto)`

Functions

- `dbl_ext_send (dbl_channel_t ch, const void *buf, size_t paylen, int flags, int *nbytes)`
  
  send on a channel and report number of bytes sent

- `dbl_ext_accept (dbl_channel_t ch, struct sockaddr *sad, int *len, void *rcontext, dbl_channel_t *rch)`

  Accept an incoming TCP connection, returns a new channel.

- `dbl_ext_list (dbl_channel_t ch)`

  Allow for incoming connections/channels.

- `dbl_ext_recv (dbl_channel_t ch, enum dbl_recvmode mode, void *buf, size_t len, struct dbl_recv_info *info)`

  Receive data from a specific TCP channel.

- `dbl_ext_recvmsg (dbl_device_t dev, enum dbl_recvmode recv_mode, struct dbl_recv_info **info, int recv-max)`

  Receive data from many channels from a same device.

- `dbl_ext_poll (dbl_channel_t *chs, int nchs, int timeout)`

  Returns number of DBL channels with pending data.

- `dbl_ext_getchopt (dbl_channel_t ch, int level, int optname, void *optval, socklen_t *optlen)`

  DBL channels are using the same option semantics than in traditional socket environment.

- `dbl_ext_setchopt (dbl_channel_t ch, int level, int optname, const void *optval, socklen_t optlen)`

  DBL channels are using the same option semantics than in traditional socket environment.

- `dbl_ext_channel_type (dbl_channel_t ch)`

  On a given channel TRUE is returned if the channel is TCP.

5.5.1 Detailed Description

API extensions for DBL.
5.5.2 Introduction to extensions

5.5.3 Function Documentation

5.5.3.1 dbl_ext_accept ( dbl_channel_t ch, struct sockaddr * sad, int * len, void * rcontext, dbl_channel_t * rch )

Accept an incoming TCP connection, returns a new channel.

Accepting incoming TCP channel connection demand.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>The channel (from dbl_bind()) on which connections are accepted</td>
</tr>
<tr>
<td>sad</td>
<td>The argument sad is a pointer to a sockaddr structure. This structure is filled with the address of the peer socket, as known to the communications layer. When addr is NULL, addrlen is not used, and should also be NULL.</td>
</tr>
<tr>
<td>len</td>
<td>The len argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by sad; on return it will contain the actual size of the peer address.</td>
</tr>
<tr>
<td>rcontext</td>
<td>The value of rcontext is associated with the new channel</td>
</tr>
<tr>
<td>rch</td>
<td>The channel which can be used to communicate with the remote peer.</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>?</td>
<td>Other codes indicate various OS failures.</td>
</tr>
</tbody>
</table>

5.5.3.2 dbl_ext_channel_type ( dbl_channel_t ch )

On a given channel TRUE is returned if the channel is TCP.

This call returns a bool on whether a channel is TCP or not.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>A valid channel</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Channel is TCP</td>
</tr>
<tr>
<td>0</td>
<td>Otherwise</td>
</tr>
</tbody>
</table>

5.5.3.3 dbl_ext_getchopt ( dbl_channel_t ch, int level, int optname, void * optval, socklen_t * optlen )

DBL channels are using the same option semantics than in traditional socket environment.

This call is used to get information on DBLTCP channel options

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>The channel</td>
</tr>
</tbody>
</table>
### Myricom DBL™

#### Level of the option (IPPROTO_IP...)
- **level**
- **optname**
- **optval**
- **optlen**

#### Return values

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>0 Success</td>
</tr>
<tr>
<td>&gt;</td>
<td>0 OS return code</td>
</tr>
</tbody>
</table>

#### Remarks

DBL channel cannot be modified or any option read. A EOPNOTSUPP return code is given back to the user in that case.

#### 5.5.3.4 dbl_ext_liste (dbl_channel_t ch )

Allow for incoming connections/channels.
Used to transition the channel into the listening state

#### Parameters

| ch | The channel (from dbl_bind()) |

#### Return values

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>?</td>
<td>Other codes indicate various OS failures.</td>
</tr>
</tbody>
</table>

#### 5.5.3.5 dbl_ext_poll (dbl_channel_t * chs, int nchs, int timeout )

Returns number of DBL channels with pending data.
Polling function for individual channels, timeout in mseconds

#### Parameters

| chs | An array of channels to query. Updated with ‘ready’ channels starting from first entry. |
| nchs | number of entries in the array |
| timeout | a timeout in milliseconds, -1 for INFINITE |

#### Remarks

An application has to pass in valid channels. For the benefit of performance, there is no error checking.

#### Return values

| number | of channels with data. Associated and updated channel array. |
5.5.3.6  
**dbl_ext_recv ( dbl_channel_t ch, enum dbl_recvmode mode, void * buf, size_t len, struct dbl_recv_info * info )**

Receive data from a specific TCP channel.

Used to check for and read data from a TCP Channel

**Parameters**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ch</strong></td>
<td>The channel (from <code>dbl_bind()</code>) on which a packet has been received.</td>
</tr>
<tr>
<td><strong>mode</strong></td>
<td>See <code>dbl_recvmode</code></td>
</tr>
<tr>
<td><strong>buf</strong></td>
<td>Buffer in which to place received data.</td>
</tr>
<tr>
<td><strong>len</strong></td>
<td>Maximum number of bytes to write into buf.</td>
</tr>
<tr>
<td><strong>info</strong></td>
<td>See <code>dbl_recv_info</code>.</td>
</tr>
</tbody>
</table>

**Return values**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td><strong>EAGAIN</strong></td>
<td>Returned if using mode <code>DBL_RECV_NONBLOCK</code> or <code>DBL_RECV_PEEK</code> when no packet is available.</td>
</tr>
<tr>
<td>?</td>
<td>Other codes indicate various OS failures.</td>
</tr>
</tbody>
</table>

**Remarks**

Receiving a return value of 0 with a msg_len of 0 means the channel is disconnected.

5.5.3.7  
**dbl_ext_recvmsg ( dbl_device_t dev, enum dbl_recvmode recv_mode, struct dbl_recv_info ** info, int recvmax )**

Receive data from many channels from a same device.

Is the extension of a `recvfrom`, but to load an array of receive information

**Parameters**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dev</strong></td>
<td>The device</td>
</tr>
<tr>
<td><strong>recv_mode</strong></td>
<td>See <code>dbl_recvmode</code></td>
</tr>
<tr>
<td><strong>info</strong></td>
<td>The array which describes in/out parameters. The important parameters are: the void *unused field used to provide the pointer to the buffer where the data should be copied, the msg_len is an input-output param, describing then len of the buffer in input, and returning the len of the message copied (see <code>dbl_recv_info</code></td>
</tr>
<tr>
<td><strong>recvmax</strong></td>
<td>the number of message which can be loaded</td>
</tr>
</tbody>
</table>

**Return values**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;= 0</td>
<td>number of messages to retrieve in the info array</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>error should be retrieved in errno</td>
</tr>
</tbody>
</table>

**Remarks**

Receiving a msg_len of 0 in the receive info structure means the channel returned is disconnected.
5.5.3.8  dbl_ext_send ( dbl_channel_t ch, const void ∗ buf, size_t paylen, int flags, int ∗ nbytes )

send on a channel and report number of bytes sent
send on DBL extension channel

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>The connected channel</td>
</tr>
<tr>
<td>buf</td>
<td>pointer to buffer</td>
</tr>
<tr>
<td>paylen</td>
<td>size to send See Flags for dbl_send(). return the number of bytes sent</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>?</td>
<td>Other codes indicate various OS failures.</td>
</tr>
</tbody>
</table>

5.5.3.9  dbl_ext_setchopt ( dbl_channel_t ch, int level, int optname, const void ∗ optval, socklen_t optlen )

DBL channels are using the same option semantics than in traditional socket environment.
This call is used to set information on DBLTCP channel options

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>The channel</td>
</tr>
<tr>
<td>level</td>
<td>Level of the option (IPPROTO_IP...)</td>
</tr>
<tr>
<td>optname</td>
<td>Option’s name (IP_TTL...)</td>
</tr>
<tr>
<td>optval</td>
<td>The pointer on the value</td>
</tr>
<tr>
<td>optlen</td>
<td>The option’s type length</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>0 Success</td>
</tr>
<tr>
<td>&gt;</td>
<td>0 OS return code</td>
</tr>
</tbody>
</table>

Remarks

DBL channel can not be modified or any option read. A EOPNOTSUPP return code is given back to the user in that case.
Chapter 6

Namespace Documentation

6.1 dbl Namespace Reference

6.1.1 Detailed Description

DBL

Author

   Myricom, Inc.
Chapter 7

Data Structure Documentation

7.1  dbl_device_attrs Struct Reference

Data Fields

• uint32_t recvq_filter_mode
• uint32_t recvq_size
• uint32_t hw_timestamping
• uint32_t reserved_1

7.1.1  Detailed Description

Structure for retrieving and setting device attributes when dbl_open is opened with DBL_OPEN_DISABLED.

7.1.2  Field Documentation

7.1.2.1  uint32_t dbl_device_attrs::hw_timestamping

Timestamp field is filled in for dbl_recv_info

7.1.2.2  uint32_t dbl_device_attrs::recvq_filter_mode

DBL receive filter mode, see dbl_filter_mode

7.1.2.3  uint32_t dbl_device_attrs::recvq_size

Host receive queue size for device
7.2  dbl_recv_info Struct Reference

Information about the packet received.

Data Fields

- `dbl_channel_t chan`
- `void * chan_context`
- `void * in_buffer`
- `struct sockaddr_in sin_from`
- `struct sockaddr_in sin_to`
- `uint32_t msg_len`
- `uint64_t timestamp`

7.2.1  Detailed Description

Information about the packet received.

7.2.2  Field Documentation

7.2.2.1  `dbl_channel_t dbl_recv_info::chan`

The channel (from `dbl_bind()`) on which a packet has been received

7.2.2.2  `void * dbl_recv_info::chan_context`

The context value passed to `dbl_bind()` when a receiving channel was created.

7.2.2.3  `void * dbl_recv_info::in_buffer`

The `in_buffer` is used in the extension of the DBL API to provide memory references in the `dbl_recvmsg()` function.

7.2.2.4  `uint32_t dbl_recv_info::msg_len`

The actual transmitted length of the packet. This may be greater than the number of bytes received if the length parameter is less than the actual number of bytes in the packet. In the case of the DBL TCP API, `msg_len` is an in-out parameter, used to fetch messages and given back to the user to indicate the length of the received packet.

7.2.2.5  `struct sockaddr_in dbl_recv_info::sin_from`

Source address of the received packet
7.2.2.6 *struct sockaddr* in *dbl_recv_info::sin_to*

Destination address of the received packet. This can be used to differentiate between packets to different multicast joins on the same channel.

7.2.2.7 *uint64_t* in *dbl_recv_info::timestamp*

Timestamp in nanosecs when the packet was received by the adapter. Timestamping must have been enabled through *dbl_device_set_attr*

### 7.3 *dbl_ticks_* Struct Reference

**Data Fields**

- `uint64_t nic_ticks`
- `uint64_t host_nsecs`
- `uint64_t host_nsecs_delay`
### Index

<table>
<thead>
<tr>
<th>API Reference</th>
<th>DBL_RECV_BLOCK, 13</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DBL_RECV_DEFAULT, 13</td>
</tr>
<tr>
<td></td>
<td>DBL_RECV_NONBLOCK, 13</td>
</tr>
<tr>
<td></td>
<td>DBL_RECV_PEEK, 13</td>
</tr>
<tr>
<td></td>
<td>DBL_RECV_PEEK_MSG, 14</td>
</tr>
<tr>
<td>API Reference, 11</td>
<td>DBL_VERSION_API, 13</td>
</tr>
<tr>
<td></td>
<td>dbl_bind, 14</td>
</tr>
<tr>
<td></td>
<td>dbl_bind_addr, 14</td>
</tr>
<tr>
<td></td>
<td>dbl_close, 15</td>
</tr>
<tr>
<td></td>
<td>dbl_device_enable, 15</td>
</tr>
<tr>
<td></td>
<td>dbl_device_get_attr, 15</td>
</tr>
<tr>
<td></td>
<td>dbl_device_handle, 16</td>
</tr>
<tr>
<td></td>
<td>dbl_device_set_attr, 16</td>
</tr>
<tr>
<td></td>
<td>dbl_filter_mode, 13</td>
</tr>
<tr>
<td></td>
<td>dbl_get_address, 16</td>
</tr>
<tr>
<td></td>
<td>dbl_getticks, 17</td>
</tr>
<tr>
<td></td>
<td>dbl_init, 17</td>
</tr>
<tr>
<td></td>
<td>dbl_mcast_block_source, 17</td>
</tr>
<tr>
<td></td>
<td>dbl_mcast_join, 18</td>
</tr>
<tr>
<td></td>
<td>dbl_mcast_join_source, 18</td>
</tr>
<tr>
<td></td>
<td>dbl_mcast_leave, 18</td>
</tr>
<tr>
<td></td>
<td>dbl_mcast_leave_source, 19</td>
</tr>
<tr>
<td></td>
<td>dbl_mcast_unblock_source, 19</td>
</tr>
<tr>
<td></td>
<td>dbl_open, 20</td>
</tr>
<tr>
<td></td>
<td>dbl_open_if, 20</td>
</tr>
<tr>
<td></td>
<td>dbl_recvfrom, 21</td>
</tr>
<tr>
<td></td>
<td>dbl_recvmode, 13</td>
</tr>
<tr>
<td></td>
<td>dbl_send, 21</td>
</tr>
<tr>
<td></td>
<td>dbl_send_connect, 22</td>
</tr>
<tr>
<td></td>
<td>dbl_send_disconnect, 22</td>
</tr>
<tr>
<td></td>
<td>dbl_sendto, 23</td>
</tr>
<tr>
<td></td>
<td>dbl_set_filter_mode, 23</td>
</tr>
<tr>
<td></td>
<td>dbl_shutdown, 23</td>
</tr>
<tr>
<td></td>
<td>dbl_unbind, 24</td>
</tr>
<tr>
<td>chan</td>
<td>dbl_recv_info, 36</td>
</tr>
<tr>
<td>chan_context</td>
<td>dbl_recv_info, 36</td>
</tr>
</tbody>
</table>
dbl_ext_channel_type
   Extensions, 29
dbl_ext_getchopt
   Extensions, 29
dbl_ext_listen
   Extensions, 30
dbl_ext_poll
   Extensions, 30
dbl_ext_recv
   Extensions, 30
dbl_ext_recvmsg
   Extensions, 31
dbl_ext_send
   Extensions, 31
dbl_ext_setchopt
   Extensions, 32
dbl_filter_mode
   API Reference, 13
dbl_getaddress
   API Reference, 16
dbl_getticks
   API Reference, 17
dbl_init
   API Reference, 17
dbl_mcast_block_source
   API Reference, 17
dbl_mcast_join
   API Reference, 18
dbl_mcast_join_source
   API Reference, 18
dbl_mcast_leave
   API Reference, 18
dbl_mcast_leave_source
   API Reference, 19
dbl_mcast_unblock_source
   API Reference, 19
dbl_open
   API Reference, 20
dbl_open_if
   API Reference, 20
dbl_recv_info, 36
   chan, 36
   chan_context, 36
   in_buffer, 36
   msg_len, 36
   sin_from, 36
   sin_to, 36
   timestamp, 37
dbl_recvfrom
   API Reference, 21
dbl_recvmode
   API Reference, 13
dbl_send
   API Reference, 21
dbl_send_connect
   API Reference, 22
dbl_send_disconnect
   API Reference, 22
dbl_sendto
   API Reference, 23
dbl_set_filter_mode
   API Reference, 23
dbl_shutdown
   API Reference, 23
dbl_ticks_, 37
dbl_unbind
   API Reference, 24
Extensions, 28
   dbl_ext_accept, 29
   dbl_ext_channel_type, 29
   dbl_ext_getchopt, 29
   dbl_ext_listen, 30
   dbl_ext_poll, 30
   dbl_ext_recv, 30
   dbl_ext_recvmsg, 31
   dbl_ext_send, 31
   dbl_ext_setchopt, 32
Flags for dbl_send(), 27
   DBL_NONBLOCK, 27
Flags used for dbl_bind(), 26
Flags used for dbl_open(), 25
   DBL_OPEN_DISABLED, 25
hw_timestamping
   dbl_device_attrs, 35
in_buffer
   dbl_recv_info, 36
msg_len
   dbl_recv_info, 36
recvq_filter_mode
   dbl_device_attrs, 35
recvq_size
   dbl_device_attrs, 35
sin_from
   dbl_recv_info, 36
sin_to
dbl_recv_info, 36

timestamp
    dbl_recv_info, 37