# Contents

1 DBL 1
   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
   1.4 Direct Access to Ring Contents ........................................ 3
   1.5 Batch Receive .............................................................. 4
   1.6 Arbitration duplicate packet elimination .............................. 4

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 ................................... 14
         5.1.4.1 dbl_ext_recvmode ............................................. 14
         5.1.4.2 dbl_filter_mode ............................................. 14
         5.1.4.3 dbl_recvmode ................................................. 14
      5.1.5 Function Documentation ............................................. 14
         5.1.5.1 dbl_ab_get_stream ......................................... 14
         5.1.5.2 dbl_ab_get_unit ............................................ 15
         5.1.5.3 dbl_ab_get_unit_protocol_info ............................ 15
         5.1.5.4 dbl_ab_set_seq ............................................. 16
         5.1.5.5 dbl_bind .................................................... 16
         5.1.5.6 dbl_bind_addr .............................................. 17
         5.1.5.7 dbl_close .................................................. 17
         5.1.5.8 dbl_device_enable ......................................... 17
         5.1.5.9 dbl_device_get_attrs ..................................... 18
         5.1.5.10 dbl_device_handle ....................................... 18
5.1.5.11 dbl_device_set_attrs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
5.1.5.12 dbl_ext_recvfrom . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
5.1.5.13 dbl_get_params . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
5.1.5.14 dbl_getaddress . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
5.1.5.15 dbl_getticks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
5.1.5.16 dbl_gettime . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
5.1.5.17 dbl_init . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
5.1.5.18 dbl_mcast_block_source . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
5.1.5.19 dbl_mcast_join . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
5.1.5.20 dbl_mcast_join_source . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
5.1.5.21 dbl_mcast_leave . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
5.1.5.22 dbl_mcast_leave_source . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.1.5.23 dbl_mcast_unblock_source . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.1.5.24 dbl_open . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.1.5.25 dbl_open_if . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
5.1.5.26 dbl_recvfrom . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
5.1.5.27 dbl_send . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
5.1.5.28 dbl_send_connect . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.1.5.29 dbl_send_disconnect . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.1.5.30 dbl_sendto . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
5.1.5.31 dbl_set_filter_mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
5.1.5.32 dbl_shutdown . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
5.1.5.33 dbl_unbind . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

5.2 Flags used for dbl_open() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2.2 Macro Definition Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2.2.1 DBL_OPEN_DISABLED . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2.2.2 DBL_OPEN_HW_TIMESTAMPING . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2.2.3 DBL_OPEN_RAW_MODE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2.2.4 DBL_OPEN_THREADSAFE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

5.3 Flags used for dbl_bind() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.3.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.3.2 Macro Definition Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.3.2.1 DBL_BIND_BROADCAST . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.3.2.2 DBL_BIND_DUP_TO_KERNEL . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.3.2.3 DBL_BIND_NO_UNICAST . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.3.2.4 DBL_BIND_REUSEADDR . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.3.2.5 DBL_BIND_TX_TIMESTAMP . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.3.3 Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.3.3.1 dbl_eventq_close . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.3.3.2 dbl_eventq_consume . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.3.3.3 dbl_eventq_count . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.3.3.4 dbl_eventq_inspect . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.3.3.5 dbl_eventq_open . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.3.3.6 dbl_eventq_peek_head . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.3.3.7 dbl_eventq_peek_next . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.3.3.8 dbl_raw_send . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.3.3.9 dbl_set_filter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

5.4 Flags for dbl_send(). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.4.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
### 5.4.2 Macro Definition Documentation
- **DBL_MCAST_LOOPBACK**
- **DBL_NONBLOCK**
- **MSG_WARM**
- **MSG_WARM**

### 5.5 Extensions
- **Detailed Description**
- **Introduction to extensions**
- **Function Documentation**
  - **dbl_ext_accept**
  - **dbl_ext_channel_type**
  - **dbl_ext_getchopt**
  - **dbl_ext_listen**
  - **dbl_ext_poll**
  - **dbl_ext_recv**
  - **dbl_ext_recvmsg**
  - **dbl_ext_send**
  - **dbl_ext_setopt**

### 5.6 Specific Options for **dbl_ext_setopt()**
- **Detailed Description**
- **Macro Definition Documentation**
  - **SO_TIMESTAMPING**

### 6 Namespace Documentation
- **dbl Namespace Reference**
  - **Detailed Description**

### 7 Data Structure Documentation
- **dbl_packet Struct Reference**
- **dbl_device_attrs Struct Reference**
  - **Detailed Description**
  - **Field Documentation**
    - **hw_timestamping**
    - **recvq_filter_mode**
    - **recvq_size**
- **dbl_ext_recv_info Struct Reference**
  - **Detailed Description**
  - **Field Documentation**
    - **buf**
    - **chan**
    - **chan_context**
    - **msg_len**
    - **sin_from**
    - **sin_to**
    - **timestamp**
- **dbl_recv_info Struct Reference**
  - **Detailed Description**
  - **Field Documentation**
    - **chan**
    - **chan_context**
    - **in_buffer**
### 7.4.2.4 msg_len
- 48
### 7.4.2.5 sin_from
- 48
### 7.4.2.6 sin_to
- 48
### 7.4.2.7 timestamp
- 48

#### 7.5 dbl_ticks_Struct Reference
- 48
#### 7.6 dbl_timespec Struct Reference
- 49
#### 7.7 unit_protocol_info Struct Reference
- 49

#### 7.7.1 Detailed Description
- 49

---

Index

50
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.

```c
dbl_init();
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.

```c
dbl_init();
dbl_open(interface, flags, &dev);
dbl_bind(dev, port1, flags, context1, &chan1);
dbl_bind(dev, port2, flags, context2, &chan2);
while (!done) {
    dbl_recvfrom(dev, mode, buf, maxlen, &info);
    user_packet_handler(buf, info.msg_len, info.chan_context);
}
```

The basic send function is 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();

```c
dbl_init();
dbl_open(interface, flags, &dev);
dbl_bind(dev, port1, flags, context1, &chan1);
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.

```c
dbl_init();
dbl_open(interface, flags, &dev);
dbl_bind(dev, port1, flags, context1, &chan1);
dbl_send_connect(chan1, address, flags, ttl, &send_handle);
dbl_send(send_handle, buf, buflen, flags);
```

To receive multicast packets, a channel joins the multicast group via dbl_mcast_join().

```c
dbl_init();
dbl_open(interface, flags, &dev);
dbl_bind(dev, port1, flags, context1, &chan1);
dbl_mcast_join(chan1, mcast_addr1, NULL);
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.

```c
dbl_init();
dbl_open(interface, flags, &dev);
dbl_bind(dev, port1, flags, context1, &chan1);
dbl_bind(dev, port2, flags, context2, &chan2);
dbl_mcast_join(chan1, mcast_addr1, NULL);
dbl_mcast_join(chan1, mcast_addr2, NULL);
dbl_mcast_join(chan2, mcast_addr1, NULL);
dbl_mcast_join(chan2, mcast_addr2, NULL);
dbl_mcast_join(chan2, mcast_addr3, NULL);
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_REUSEADDR 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.

1.4 Direct Access to Ring Contents

The DBL API allows an application to directly access the packet buffers. An application can peek at a packet without consuming it. For example, a trading application could peek at the packet to search for a specific sequence or symbol. The application can then consume packets, without overhead, until a meaningful sequence or symbol is found, which may improve overall latency.

An application can operate on ring data without copying the payload. The API has functions to get a pointer to the first and next packets in the queue and pointers to the header and data sections of a given packet. There is also a function to consume the packet using two modes: DBL_CONSUME_SINGLE, which consumes a single packet at the head of the queue; and DBL_CONSUME_ALL, which consumes all outstanding packets. See the following functions

dbl_eventq_open dbl_eventq_close dbl_eventq_peek_head dbl_eventq_peek_next dbl_eventq_inspect dbl_eventq_consume
See the example programs in the bin/tests directory and described in the DBL User Guide: `dbl_ring_access` and `dbl_eventq`.

### 1.5 Batch Receive

The DBL API allows an application to determine if further packets are pending. The API also provides a function `dbl_ext_recvfrom` that allows you to receive multiple packets in one call instead of receiving packets sequentially, one at a time `dbl_recvfrom`, minimizing overhead. See the following function: `dbl_ext_recvfrom`.

See the example program in the bin/tests directory and described in the DBL User Guide: `dbl_batch_recv`.

### 1.6 Arbitration duplicate packet elimination

Financial exchanges often use redundant multicast UDP feeds to deceminate pricing information. Since UDP is unreliable, the redundancy is used to reduce loss of information.

The application or middleware is normally responsible for eliminating the redundant packets and providing a single more reliable stream of packets to the trading engine. This process is coined "AB Arbitration".

While the path of UDP packets from the Exchange to the Phoenix is unreliable, the path from Phoenix to application is reliable, so it is appropriate to eliminate redundant packets within the Phoenix. Phoenix can do a best-effort optimization to eliminate redundant packets.

To enable the redundant packet elimination feature, 2 environment variables are used: `DBL_AB_PROTO_FILE` - full path name of the XML file describing packet formats used by supported Exchange feeds. `DBL_AB_FEEDS_FILE` - full path name of an XML file defining the multicast addresses and UDP ports for streams of units of an Exchange feed. See the DBL User Guide for more information.
Chapter 2

Module Index

2.1 API Reference

Here is a list of all modules:

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Reference</td>
<td>11</td>
</tr>
<tr>
<td>Flags used for dbl_open()</td>
<td>29</td>
</tr>
<tr>
<td>Flags used for dbl_bind()</td>
<td>30</td>
</tr>
<tr>
<td>Flags for dbl_send()</td>
<td>35</td>
</tr>
<tr>
<td>Extensions</td>
<td>36</td>
</tr>
<tr>
<td>Specific Options for dbl_ext_setopt()</td>
<td>41</td>
</tr>
</tbody>
</table>
Chapter 3

Namespace Index

3.1 Namespace List

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

- db1

  43
Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

- `dbl__packet` .......................................................... 45
- `dbl_device_attrs` .................................................. 45
- `dbl_ext_recv_info` .................................................... 46
  Information about the packet received ........................ 46
- `dbl_recv_info` .......................................................... 47
  Information about the packet received ........................ 47
- `dbl_ticks_` .............................................................. 48
- `dbl_timespec` .......................................................... 49
- `unit_protocol_info` .................................................. 49
Chapter 5

Module Documentation

5.1 API Reference

API Reference for DBL.

Data Structures

- `struct dbl_device_attrs`
- `struct dbl_ext_recv_info`
  
  *Information about the packet received.*
- `struct dbl_recv_info`
  
  *Information about the packet received.*
- `struct unit_protocol_info`

Modules

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

Macros

- `#define DBL_VERSION_API 0x0006`

Enumerations

- `enum dbl_filter_mode { DBL_RECV_FILTER_NORMAL = 0, DBL_RECV_FILTER_ALLMULTI = 1, DBL_RECV_FILTER_RAW = 2 }`
- `enum dbl_ext_recvmode { DBL_EXT_RECV_DEFAULT = 0, DBL_EXT_RECV_NONBLOCK = 1, DBL_EXT_RECV_COMPLETE = 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, DBL_RECV_TX_TIMESTAMP = 6, DBL_RECV_DEFAULT_RAW = 7
}

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)`
  Returns a descriptor for use with `poll()` or `select()`.

- `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 dep, enum dbl_filter_mode mode)`

- `dbl_device_handle (dbl_device_t dev)`

- `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_gettime (dbl_device_t dev, dbl_timespec_t *ts)`
  Retrieve the current NIC clock value.

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

- `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_get_params** (dbl_device_t dev, void *dbl_params)

  *Used to query global and local DBL settings.*

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

  *Unblock dbl_recvfrom/dbl_ext_recvmsg.*

• **dbl_ext_recvfrom** (dbl_device_t dev, enum dbl_ext_recvmode mode, int *num, struct dbl_ext_recvinfo *info)

  *Receive (UDP) data.*

• **dbl_recvfrom** (dbl_device_t dev, enum dbl_recvmode mode, void *buf, size_t len, struct dbl_recvinfo *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.*

• **dbl_ab_set_seq** (dbl_device_t dev, struct sockaddr_in *sad, int seq_num)

  *Reset the sequence number.*

• **dbl_ab_get_stream** (dbl_device_t dep, int unit_id, int abcd, struct sockaddr_in *sad)

  *Return a stream given unit_id and a/b/c/d.*

• **dbl_ab_get_unit** (dbl_device_t dev, struct sockaddr_in *sad, int *unit_id)

  *Return the unit_id.*

• **dbl_ab_get_unit_protocol_info** (dbl_device_t dev, struct sockaddr_in *sad, struct unit_protocol_info *upi)

  *Return the protocol information.*

### 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 0x0006

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_recvinfo
- 0x0003: Added TCP extensions
- 0x0004: Added tx timestamping
- 0x0005: struct dbl_recvinfo changes
- 0x0006: struct dbl_recvinfo should not have changed, revert
5.1.4 Enumeration Type Documentation

5.1.4.1 enum dbl_ext_recvmode

Specifies behavior of the `dbl_ext_recvfrom` call

**Enumerator**

- **DBL_EXT_RECV_DEFAULT**  Busy poll forever until at least one packet has been received.
- **DBL_EXT_RECV_NONBLOCK**  Return a packet if available, else return EAGAIN.
- **DBL_EXT_RECV_COMPLETE**  wait until all provided buffers are filled.

5.1.4.2 enum 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.

5.1.4.3 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
- **DBL_RECV_TX_TIMESTAMP**  nonblocking recv for retrieving full payloads and associated timestamps for outgoing (tx) traffic, restricted to `dbl_ext` functions only, EINVAL reported otherwise
- **DBL_RECV_DEFAULT_RAW**  blocking recv for retrieving full headers and payloads and associated timestamps for incoming (rx) traffic.

5.1.5 Function Documentation

5.1.5.1 `dbl_ab_get_stream ( dbl_device_t dep, int unit_id, int abcd, struct sockaddr_in *sad )`

Return a stream given `unit_id` and `abcd`.
Return a stream pointer given a unique A/B arbitration `unit_id` and stream.

**Parameters**
The device

The unique id assigned to this feed unit when parsed from the feeds file.

The positional definition of the stream in the feeds file.

The argument sad is a pointer to a sockaddr structure. This structure is filled with stream address information upon return.

Return values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EPERM operation not permitted because AB is not enabled</td>
<td></td>
</tr>
<tr>
<td>EINVAL unit_id not found or abcd &gt; number of streams for this unit</td>
<td></td>
</tr>
</tbody>
</table>

5.1.5.2 dbl_ab_get_unit ( dbl_device_t dev, struct sockaddr_in *sad, int *unit_id )

Return the unit_id.

Return the unique unit ID associated with a multicast socket address.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>The device</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 any of the streams which may be using the A/B arbiter.</td>
</tr>
<tr>
<td>unit_id</td>
<td>The unique identifier will be returned in this parameter</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>EINVAL the sockaddr pointer was NULL</td>
<td></td>
</tr>
<tr>
<td>EPERM operation not permitted because AB is not enabled</td>
<td></td>
</tr>
</tbody>
</table>

5.1.5.3 dbl_ab_get_unit_protocol_info ( dbl_device_t dev, struct sockaddr_in *sad, struct unit_protocol_info *upi )

Return the protocol information.

Return protocol information associated with the given multicast socket address.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>The device</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 any of the streams which may be using the A/B arbiter.</td>
</tr>
<tr>
<td>upi</td>
<td>Protocol information associated with provided multicast address will be returned here</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>EINVAL the sockaddr pointer was NULL</td>
<td></td>
</tr>
<tr>
<td>EPERM operation not permitted because AB is not enabled</td>
<td></td>
</tr>
</tbody>
</table>
5.1.5.4  

dbl_ab_set_seq ( dbl_device_t dev, struct sockaddr_in * sad, int seq_num )

Reset the sequence number.
Reset the sequence number in an A/B arbitration unit.

Parameters

| dev    | The device |
| sad    | The argument sad is a pointer to a sockaddr structure. This structure is filled with the address of any of the streams using the A/B arbiter. When addr is NULL, the seq_num will be applied to all arbitration units. |
| seq_num | The seq_num is the next expected sequence number for a packet. |

Return values

| == 0 | Success |
| != 0 | OS return code |
| == EPERM | operation not permitted because AB is not enabled |

5.1.5.5  

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

| dev    | A DBL device handle returned by a call to dbl_open(). |
| flags  | See Flags used for dbl_bind(). |
| port   | The port to send/receive on. |
| context | The value of context is returned on future receives on this channel. |
| handle_out | The handle to the created channel. |

Return values

| 0 | Success |
| EINVAL | Error in arguments |
| EEXIST | port already in use |
| ? | Other values indicate various OS failures in the bind process |

If dbl_bind() 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.

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.
5.1.5.6  **dbl_bind_addr** ( db_device_t dev, const struct in_addr * ipaddr, int flags, int port, void * context, db_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

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

### 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>Error in arguments. Specifying an address that is not on the same underlying interface as that specified with <code>dbl_open</code> will return EINVAL.</td>
</tr>
<tr>
<td>EEXIST</td>
<td>port already in use</td>
</tr>
<tr>
<td>?</td>
<td>Other values indicate various OS failures in the bind process</td>
</tr>
</tbody>
</table>

### Remarks

DBL TCP supported

5.1.5.7  **dbl_close** ( db_device_t dev )

Close a db device.

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

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| dev       | The device handle returned from `dbl_open()`.

### Return values

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

5.1.5.8  **dbl_device_enable** ( db_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.

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

Function to retrieve device attributes.

Parameters

<table>
<thead>
<tr>
<th>dev</th>
<th>The device handle returned from <code>dbl_open()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>Device attributes will be copied out.</td>
</tr>
</tbody>
</table>

Remarks

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

5.1.5.10 `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

5.1.5.11 `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

<table>
<thead>
<tr>
<th>dev</th>
<th>The device handle returned from <code>dbl_open()</code> with flag <code>DBL_OPEN_DISABLED</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>Device attributes that will be set on the device.</td>
</tr>
</tbody>
</table>

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 accomodate internal alignment and sizing requirements. If these sizes are changed, the new sizes are reflected during a subsequent call to `dbl_device_get_attrs`. 
Receive (UDP) data. Used to check for and read data from the channels associated with a particular dbl_device.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dev</code></td>
<td>The underlying device via <code>dbl_open</code></td>
</tr>
<tr>
<td><code>mode</code></td>
<td>See <code>dbl_ext_recvmode</code></td>
</tr>
<tr>
<td><code>num</code></td>
<td>In / Out value of given buffers pointed by <code>struct dbl_ext_recv_info</code> and buffers detected via underlying receive operations.</td>
</tr>
<tr>
<td><code>info</code></td>
<td>A pointer to an array. See <code>dbl_ext_recv_info</code>.</td>
</tr>
</tbody>
</table>

### Return values

- **0**: Success
- **EAGAIN**: Returned if using mode `DBL_RECV_NONBLOCK` or `DBL_RECV_COMPLETE` when no packet is available.
- **EINVAL**: in case `dbl_shutdown()` was called
- **ENOBUS**: provided buffer was filled successfully but more data is pending.
- **?**: Other codes indicate various OS failures.

### Remarks

`dbl_ext_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. DBL TCP not supported.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dev</code></td>
<td>The underlying device via <code>dbl_open</code></td>
</tr>
<tr>
<td><code>params</code></td>
<td>struct pointer</td>
</tr>
</tbody>
</table>

### Remarks

DBL UDP and TCP

### Return values

- **0**: Success, parameter len updated with actual length

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ch</code></td>
<td>A channel to query</td>
</tr>
<tr>
<td><code>sin</code></td>
<td>A socket address</td>
</tr>
</tbody>
</table>

###Remarks

Returns the address to which a channel is bound.
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

5.1.5.15  **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**

| dev | Specifies the dev channel from dbl_open |
| ticks | Specifies the dbl_ticks_t structure holding the timing information |

**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 dev specified</td>
</tr>
</tbody>
</table>

**Remarks**

DBL TCP supported
Under TA, a ioclt/WSAIoctl socket call can use cmd SIO_GETNICTIME

5.1.5.16  **dbl_gettime ( dbl_device_t dev, dbl_timespec_t *ts )**

Retrieve the current NIC clock value.

Retrieves the current NIC clock value by reading the timestamp register on the NIC.

**Parameters**

| dev | The dev channel from dbl_open. |
| ts | dbl_timespec_t that will hold the clock value upon return. |

**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>Argument error such as bad dev.</td>
</tr>
</tbody>
</table>
Remarks

Reading the timestamp register is a blocking read operation that goes over local interconnect and may have non-trivial delays on the order of 1 microsecond. This method does not adjust the clock value read from the NIC in any way.

5.1.5.17  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

<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/incompatible version passed.</td>
</tr>
</tbody>
</table>

Remarks

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

5.1.5.18  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

- **ch**: Handle for the channel to leave the specified multicast group.
- **join_addr**: Address of the multicast group to join.
- **block_addr**: Address to block. The multicast packets will not be received from the blocked source.

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>

5.1.5.19  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><code>ch</code></td>
<td>Handle for the channel to add to the specified multicast group.</td>
</tr>
<tr>
<td><code>mcast_addr</code></td>
<td>Address of the multicast group to join.</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><code>EINVAL</code></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>

5.1.5.20  `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><code>ch</code></td>
<td>Handle for the channel to add to the specified multicast group.</td>
</tr>
<tr>
<td><code>mcast_addr</code></td>
<td>Address of the multicast group to join.</td>
</tr>
<tr>
<td><code>src</code></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><code>EINVAL</code></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>

5.1.5.21  `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><code>ch</code></td>
<td>Handle for the channel to leave the specified multicast group.</td>
</tr>
<tr>
<td><code>mcast_addr</code></td>
<td>Address of the multicast group to leave.</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><code>EINVAL</code></td>
<td>Argument error, such as address not multicast group.</td>
</tr>
<tr>
<td><code>EADDRNOTAVAIL</code></td>
<td>Not currently joined to group &quot;address&quot;</td>
</tr>
<tr>
<td><code>EAGAIN</code></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>
5.1.5.22  
```
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>ch</th>
<th>Handle for the channel to leave the specified multicast group.</th>
</tr>
</thead>
<tbody>
<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**

<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>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>

5.1.5.23  
```
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>ch</th>
<th>Handle for the channel to leave the specified multicast group.</th>
</tr>
</thead>
<tbody>
<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**

<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>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>

5.1.5.24  
```
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>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>bad usage. includes dbl_init not called first and bad interface_addr.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>no matching IP address found on DBL-enabled NIC</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.

### 5.1.5.25 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifname</td>
<td>Specifies the name of the interface with which channels created using dbl_bind() will be associated.</td>
</tr>
<tr>
<td>flags</td>
<td>A bitmask of flags to alter open behavior. See Flags used for dbl_open()</td>
</tr>
<tr>
<td>dev_out</td>
<td>On successful return, this is where the handle for the newly opened device will be placed.</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>EINVAL</td>
<td>bad usage. includes dbl_init not called first and bad interface_addr.</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.
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>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.

5.1.5.27  

dbl_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>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendh</td>
<td>Send handle specifying destination for packet.</td>
</tr>
<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>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>EAGAIN</td>
<td>SO_TIMESTAMPING option enabled and no TX timestamping queue exhausted. An application would need to retrieve TX packets via the DBL_RECV_MSG_ERRQUEUE at this time to be able to send again or disable the TX timestamping option.</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 DBL flags. The function will block until all data has been transferred. For advanced handling use dbl_ext_send for TCP channels.

Return values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EISCONN</td>
<td>channel already connected</td>
</tr>
</tbody>
</table>
5.1.5.28  

dbl_send_connect ( dbl_channel_t chan, const struct sockaddr * 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

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chan</td>
<td>The channel to be associated with this send handle.</td>
</tr>
<tr>
<td>dest_sin</td>
<td>Destination address of packets sent using this handle.</td>
</tr>
<tr>
<td>flags</td>
<td>Bitmask of flags to modify default send_connect operation Currently no flags are supported.</td>
</tr>
<tr>
<td>ttl</td>
<td>The value to put in the TTL field of the IP header.</td>
</tr>
<tr>
<td>hsend</td>
<td>The send_handle to be used in future calls to dbl_send() is returned here.</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Return value</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.

5.1.5.29  

dbl_send_disconnect ( dbl_send_t hsend )

Release a send handle.

Release the resources associated with a send handle.

Parameters

| hsend | The send handle. |

Return values

<table>
<thead>
<tr>
<th>Return 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.
5.1.5.30 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>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Handle for the channel to send over.</td>
</tr>
<tr>
<td>sin</td>
<td>The destination address</td>
</tr>
<tr>
<td>buf</td>
<td>The data to send.</td>
</tr>
<tr>
<td>len</td>
<td>The length of the data 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>

5.1.5.31 dbl_set_filter_mode (dbl_device_t dep, enum dbl_filter_mode mode)

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

5.1.5.32 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>The underlying device via dbl_open</td>
</tr>
<tr>
<td>how</td>
<td>Unused for now</td>
</tr>
</tbody>
</table>

Remarks

DBL UDP and TCP

5.1.5.33 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>Parameter</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

| 0 | Success |

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
- #define DBL_OPEN_RAW_MODE 0x8

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 timestamps are 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.

Note, starting with DBL v4, this is enabled by default.

5.2.2.3 #define DBL_OPEN_RAW_MODE 0x8

Request that incoming packets not only contain the payload but also fully qualified ethernet and IP headers. Payload then starts with an offset. An endpoint needs to be opened with DBL_OPEN_RAW_MODE to be able to use the DBL_RECV_RAW recv flag. Note that the following flags are not available when using RAW mode: DBL_RECV_PEEK, DBL_RECV_PEEK_MSG

5.2.2.4 #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
- #define DBL_BIND_TX_TIMESTAMP 0x20

Enumerations

- enum dbl_consumemode { DBL_CONSUME_SINGLE = 0, DBL_CONSUME_ALL = 1 }
- enum dbl_filtermode { DBL_ADD_FILTER = 0, DBL_REMOVE_FILTER = 1 }

Functions

- dbl_raw_send (dbl_device_t dep, uint8_t *hdr, int hdrlen, uint32_t chksum_hdr, int chksum_offset, const uint8_t *buffer, int paylen, int flags)
  Sends RAW packet on a DBL endpoint opened in RAW mode.
- dbl_eventq_open (dbl_device_t dep)
  Open dbl_device for eventq functions.
- dbl_eventq_peek_head (dbl_device_t dep, dbl_packet_t *pkt)
  Peek first packet.
- dbl_eventq_peek_next (dbl_device_t dep, dbl_packet_t pkt, dbl_packet_t *npkt)
  Peek next packet.
- dbl_eventq_inspect (dbl_device_t dev, dbl_packet_t pkt, char **hdr, char **data, int *len, uint64_t *timestamp)
  Get pointers for processing.
- dbl_eventq_consume (dbl_device_t dev, dbl_packet_t pkt, enum dbl_consumemode consume_mode)
  Consume the packet. Advance on receive queue.
- dbl_eventq_close (dbl_device_t dep)
  Close a dbl device opened by dbl_eventq_open().
- dbl_eventq_count (dbl_device_t dep)
  Count available packets ready to receive.
- dbl_set_filter (dbl_device_t dep, enum dbl_filtermode how, int ip_proto, struct sockaddr_in *dst, int flags)
  Set filters on a dbl raw endpoint.

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.

5.3.2.5 #define DBL_BIND_TX_TIMESTAMP 0x20

Allows this channel to receive tx timestamped packets. Requires DBL VERSION API 0x0004 or higher

5.3.3 Function Documentation

5.3.3.1 dbl_eventq_close ( dbl_device_t dep )

Close a dbl device opened by dbl_eventq_open().
Terminate usage of a device opened by dbl_eventq_open()

Parameters

<table>
<thead>
<tr>
<th>dev</th>
<th>The device handle returned from dbl_open().</th>
</tr>
</thead>
</table>

Return values

| 0 | Success |

5.3.3.2 dbl_eventq_consume ( dbl_device_t dev, dbl_packet_t pkt, enum dbl_consumemode consume_mode )

Consume the packet. Advance on receive queue.
Consume the packet

Parameters

| pkt | reference packet to consume. |

Returns

0 successful EAGAIN if packet is not head. It will be able to be consumed successful the next time if the pkt became head.
### 5.3.3.3 `dbl_eventq_count ( dbl_device_t dep )`

Count available packets ready to receive.
Counts available packets in recv queue but won’t be reporting any packet lengths.

**Parameters**

- `dev` | A DBL device handle returned by a call to `dbl_open()`.

**Returns**

number of pending packets.

### 5.3.3.4 `dbl_eventq_inspect ( dbl_device_t dev, dbl_packet_t pkt, char **hdr, char **data, int *len, uint64_t *timestamp )`

Get pointers for processing.
Get pointers to hdr and data for given packet

**Parameters**

- `dev` | A DBL device handle returned by a call to `dbl_open()`.
- `pkt` | reference packet for inspection
- `hdr` | (Ethernet) HDRs, can be NULL
- `data` | pointer to payload, can be NULL
- `len` | len of payload, can be NULL
- `timestamp` | return timestamp of packet, can be NULL

**Returns**

- 0 successful
- EINVAL if params don’t match

### 5.3.3.5 `dbl_eventq_open ( dbl_device_t dep )`

Open `dbl_device` for eventq functions.
Create an instance of a `dbl_device` which can be used to subsequently open channels via `dbl_bind()`.
Opens a device for use with eventq functions
Open a `dbl_device` for eventq functions.

**Parameters**

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

**Return values**

| 0 | Success |
5.3.3.6 `dbl_eventq_peek_head (dbl_device_t dep, dbl_packet_t *pkt)`

Peek first packet.

Gets pointer to head of receive queue.

**Parameters**

| dev | A DBL device handle returned by a call to `dbl_open()`.  
| pkt | for first unread packet in queue |

**Returns**

0 for success, ENOBUFS for empty queue

5.3.3.7 `dbl_eventq_peek_next (dbl_device_t dep, dbl_packet_t pkt, dbl_packet_t *npkt)`

Peek next packet.

Gets pointer to following packet

**Parameters**

| dev | A DBL device handle returned by a call to `dbl_open()`.  
| pkt | reference packet for returning next packet to it  
| npkt | references to next packet |

**Returns**

0 for success, ENOBUFS for no more packets

5.3.3.8 `dbl_raw_send (dbl_device_t dep, uint8_t *hdr, int hdrlen, uint32_t chksum_hdr, int chksum_offset, const uint8_t *buffer, int paylen, int flags)`

Sends RAW packet on a DBL endpoint opened in RAW mode.

Sends a fully qualified packet in raw format to device. When using a DBL RAW endpoint, this function expects the provided buffer containing a fully qualified ether frame.

**Parameters**

| dev | A DBL RAW device handle returned by a call to `dbl_open()`.  
| hdr | IPV4 HDR  
| hdrlen |  
| chksum_hdr | hdr checksum  
| chksum_offset | offset  
| buffer | payload  
| paylen | length  
| flags | e.g DBL_TX_LOOPBACK or DBL_MCAST_LOOPBACK |
Set filters on a dbl raw endpoint.

Sets a filter for receiving raw packets. When using a DBL RAW endpoint, this function allows for passing in a filter so that packets go up to user space.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dev</code></td>
<td>A DBL RAW device handle returned by a call to <code>dbl_open()</code>.</td>
</tr>
<tr>
<td><code>how</code></td>
<td>Add or remove filter</td>
</tr>
<tr>
<td><code>ip_proto</code></td>
<td>UDP or TCP</td>
</tr>
<tr>
<td><code>dst</code></td>
<td>Destination IP address and port</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Optional flags for filter DBL_BIND_DUP_TO_KERNEL</td>
</tr>
</tbody>
</table>

**Return values**

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td><code>EINVAL</code></td>
<td>DBL EP not opened in RAW mode</td>
</tr>
</tbody>
</table>
5.4 Flags for dbl_send().

Macros

- #define DBL_NONBLOCK 0x4
- #define MSG_WARM 0x20000
- #define MSG_WARM 0x20000
- #define DBL_MCAST_LOOPBACK 0x10
- #define DBL_TX_LOOPBACK 0x8

5.4.1 Detailed Description

5.4.2 Macro Definition Documentation

5.4.2.1 #define DBL_MCAST_LOOPBACK 0x10

Loop back mcast data to local host (default off)

5.4.2.2 #define DBL_NONBLOCK 0x4

Return EAGAIN if send request would block for resources

5.4.2.3 #define MSG_WARM 0x20000

Optimize send() cost: Keep TCP send path warm. Data is not put on the wire.

5.4.2.4 #define MSG_WARM 0x20000

Optimize send() cost: Keep TCP send path warm. Data is not put on the wire.
5.5 Extensions

API extensions for DBL.

Modules

- Specific Options for dbl_ext_setchopt().

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 MSG_ERRQUEUE 0x2000`
- `#define DBL_CHANNEL_FLAGS(type, proto) DBL_INITFLAGS(type, proto)`

Enumerations

- `enum { SO_TX_TIMESTAMPING = 0x0900 }

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_listeen (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 non-DBL 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)`
  Report on available incoming data on DBL channels.
- `dbl_ext_setchopt (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></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ch</strong></td>
<td>The channel (from <strong>dbl_bind</strong>) on which connections are accepted</td>
</tr>
<tr>
<td><strong>sad</strong></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><strong>len</strong></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><strong>rcontext</strong></td>
<td>The value of rcontext is associated with the new channel</td>
</tr>
<tr>
<td><strong>rch</strong></td>
<td>The channel which can be used to communicate with the remote peer.</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>?</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></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ch</strong></td>
<td>A valid channel</td>
</tr>
</tbody>
</table>

**Return values**

<table>
<thead>
<tr>
<th></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_getopt (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**

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

**Return values**

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

### 5.5.3.4 `dbl_listen (dbl_channel_t ch)`

Allow for incoming connections/channels.

Used to transition the channel into the listening state.

**Parameters**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ch</code></td>
<td>The channel (from <code>dbl_bind()</code>)</td>
</tr>
</tbody>
</table>

**Return values**

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

### 5.5.3.5 `dbl_poll (dbl_channel_t *chs, int nchs, int timeout)`

Report on available incoming data on DBL channels.

Polling function for multiplexed channels, in/out channels in `chs`, timeout in milliseconds.

**Parameters**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chs</code></td>
<td>An array of channels to query</td>
</tr>
<tr>
<td><code>nchs</code></td>
<td>Number of entries in the array</td>
</tr>
<tr>
<td><code>timeout</code></td>
<td>A timeout in milliseconds, -1 for INFINITE</td>
</tr>
</tbody>
</table>
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 non-DBL channel.
Used to check for and read data from a specific channel if non-DBL channel

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

<table>
<thead>
<tr>
<th>Return values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EAGAIN</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

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

<table>
<thead>
<tr>
<th>Return values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;= 0</td>
<td>number of messages to retrieve in the <code>info</code> array</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>error should be retrieved in <code>errno</code></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></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></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...) and specific options, see Specific Options for dbl_ext_setchopt().</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.
5.6 Specific Options for dbl_ext_setchopt().

Mcros

- #define SO_TIMESTAMPING 0x0025

5.6.1 Detailed Description

5.6.2 Macro Definition Documentation

5.6.2.1 #define SO_TIMESTAMPING 0x0025

Enable given channel to perform tx timestamping when sending. Retrieve timestamps via DBL_MSG_ERRQUEUE.
Chapter 6

Namespace Documentation

6.1  dbl Namespace Reference

6.1.1  Detailed Description

DBL

Author
  CSP Inc.
Chapter 7

Data Structure Documentation

7.1 dbl__packet Struct Reference

Data Fields

- struct dbl__ep * dep
- uintptr_t uevt
- uintptr_t desc_offset
- uintptr_t data_offset
- uint64_t length
- void * pkt
- void * hdr
- void * payload
- int paylen
- uint64_t timestamp

7.2 dbl_device attrs Struct Reference

Data Fields

- uint32_t recvq_filter_mode
- uint32_t recvq_size
- uint32_t hw_timestamping
- uint32_t reserved_1

7.2.1 Detailed Description

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

7.2.2.1 uint32_t dbl_device_attrs::hw_timestamping

Timestamp field is filled in for dbl_recv_info

7.2.2.2 uint32_t dbl_device attrs::recvq_filter_mode

DBL receive filter mode, see dbl_filter_mode

7.2.2.3 uint32_t dbl_device attrs::recvq_size

Host receive queue size for device

7.3 dbl_ext_recv_info Struct Reference

Information about the packet received.

Data Fields

- dbl_channel_t chan
- void * chan_context
- void * buf
- int buflen
- struct sockaddr_in sin_from
- struct sockaddr_in sin_to
- uint32_t msg_len
- uint64_t timestamp

7.3.1 Detailed Description

Information about the packet received.

7.3.2 Field Documentation

7.3.2.1 void* dbl_ext_recv_info::buf

The buffer is to be used for data

7.3.2.2 dbl_channel_t dbl_ext_recv_info::chan

The channel (from dbl_bind()) on which a packet has been received
The context value passed to `dbl_bind()` when a receiving channel was created.

The actual transmitted length of the packet.

Source address of the received packet

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

Timestamp in nanosecs when the packet was received by the adapter.

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`

Information about the packet received.

Information about the packet received.
7.4.2 Field Documentation

7.4.2.1 `dbl_channel_t dbl_recv_info::chan`

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

7.4.2.2 `void* dbl_recv_info::chan_context`

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

7.4.2.3 `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.

7.4.2.4 `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.

7.4.2.5 `struct sockaddr_in dbl_recv_info::sin_from`

Source address of the received packet.

7.4.2.6 `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.

7.4.2.7 `uint64_t 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`.

7.5 `dbl_ticks` Struct Reference

Data Fields

- `uint64_t nic_ticks`
- `uint64_t host_nsecs`
- `uint64_t host_nsecs_delay`
7.6  dbl_timespec Struct Reference

Data Fields

- long tv_sec
- long tv_nsec

7.7  unit_protocol_info Struct Reference

Data Fields

- uint8_t header_offset
- uint8_t header_length
- uint8_t header_big_endian
- uint8_t sequence_offset
- uint8_t sequence_length
- uint8_t sequence_big_endian
- uint8_t message_count_offset
- uint8_t message_count_length
- uint8_t message_count_big_endian

7.7.1 Detailed Description

This structure is used to transfer internal unit protocol information to the user when `dbl_ab_get_unit_protocol_info` is called.
Index

API Reference
DBL_EXT_RECV_COMPLETE, 14
DBL_EXT_RECV_DEFAULT, 14
DBL_EXT_RECV_NONBLOCK, 14
DBL_RECV_BLOCK, 14
DBL_RECV_DEFAULT, 14
DBL_RECV_DEFAULT_RAW, 14
DBL_RECV_NONBLOCK, 14
DBL_RECV_PEEK, 14
DBL_RECV_PEEK_MSG, 14
DBL_RECV_TX_TIMESTAMP, 14

API Reference, 11
DBL_VERSION_API, 13
dbl_ab_get_stream, 14
dbl_ab_get_unit, 15
dbl_ab_get_unit_protocol_info, 15
dbl_ab_set_seq, 15
dbl_bind, 16
dbl_bind_addr, 16
dbl_close, 17
dbl_device_enable, 17
dbl_device_get_attrs, 18
dbl_device_handle, 18
dbl_device_set_attrs, 18
dbl_ext_recvfrom, 18
dbl_ext_recvmode, 14
dbl_filter_mode, 14
dbl_get_params, 19
dbl_getaddress, 19
dbl_gettime, 20
dbl_gettimeofday, 20
dbl_init, 21
dbl_mcast_block_source, 21
dbl_mcast_join, 21
dbl_mcast_join_source, 22
dbl_mcast_leave, 22
dbl_mcast_leave_source, 22
dbl_mcast_unblock_source, 23
dbl_open, 23
dbl_open_if, 24
dbl_recvfrom, 24
dbl_recvmode, 14
dbl_send, 25
dbl_send_connect, 25
dbl_send_disconnect, 26
dbl_sendto, 26
dbl_set_filter_mode, 27
dbl_shutdown, 27
dbl_unbind, 27
buf
dbl_ext_recv_info, 46
chan
dbl_ext_recv_info, 46
dbl_recvmode, 27
dbl_recv_info, 48
chan_context
dbl_ext_recv_info, 46
dbl_recv_info, 48

DBL_EXT_RECV_COMPLETE
API Reference, 14
DBL_EXT_RECV_DEFAULT
API Reference, 14
DBL_EXT_RECV_NONBLOCK
API Reference, 14
DBL_RECV_BLOCK
API Reference, 14
DBL_RECV_DEFAULT
API Reference, 14
DBL_RECV_DEFAULT_RAW
API Reference, 14
DBL_RECV_NONBLOCK
API Reference, 14
DBL_RECV_PEEK
API Reference, 14
DBL_RECV_PEEK_MSG
API Reference, 14
DBL_RECV_TX_TIMESTAMP
API Reference, 14
DBL_BIND_BROADCAST
Flags used for dbl_bind(), 30
DBL_BIND_REUSEADDR
API Reference, 22

dbl_mcast_leave
API Reference, 22

dbl_mcast_leave_source
API Reference, 22

dbl_mcast_unblock_source
API Reference, 23

dbl_open
API Reference, 23

dbl_open_if
API Reference, 24

dbl_raw_send
Flags used for dbl_bind(), 33

dbl_recv_info, 47
  chan, 48
  chan_context, 48
  in_buffer, 48
  msg_len, 48
  sin_from, 48
  sin_to, 48
  timestamp, 48

dbl_recvfrom
API Reference, 24

dbl_recvmode
API Reference, 14

dbl_send
API Reference, 25

dbl_send_connect
API Reference, 25

dbl_send_disconnect
API Reference, 26

dbl_sendto
API Reference, 26

dbl_set_filter
Flags used for dbl_bind(), 33

dbl_set_filter_mode
API Reference, 27

dbl_shutdown
API Reference, 27

dbl_ticks_, 48

dbl_timespec, 49

dbl_unbind
API Reference, 27

Extensions, 36

dbl_ext_accept, 37

dbl_ext_channel_type, 37

dbl_ext_getchopt, 37

dbl_ext_listen, 38

dbl_ext_poll, 38

dbl_ext_recv, 38

dbl_ext_recvmsg, 39

dbl_ext_send, 39

dbl_ext_setchopt, 40

Flags for dbl_send(), 35
  DBL_NONBLOCK, 35
  MSG_WARM, 35

Flags used for dbl_bind(), 30
  dbl_eventq_close, 31
  dbl_eventq_consume, 31
  dbl_eventq_count, 31
  dbl_eventq_inspect, 32
  dbl_eventq_open, 32
  dbl_eventq_pEEK_head, 32
  dbl_eventq_pEEK_next, 33
  dbl_raw_send, 33
  dbl_set_filter, 33

Flags used for dbl_open(), 29
  DBL_OPEN_DISABLED, 29

hw_timestamping
  dbl_device_attrs, 46

in_buffer
  dbl_recv_info, 48

MSG_WARM
  Flags for dbl_send(), 35
  msg_len
    dbl_ext_recv_info, 47
    dbl_recv_info, 48

recvq_filter_mode
  dbl_device_attrs, 46

recvq_size
  dbl_device_attrs, 46

SO_TIMESTAMPING
  Specific Options for dbl_ext_setchopt(), 41

sin_from
  dbl_ext_recv_info, 47
  dbl_recv_info, 48

sin_to
  dbl_ext_recv_info, 47
  dbl_recv_info, 48

Specific Options for dbl_ext_setchopt(), 41
  SO_TIMESTAMPING, 41

timestamp
  dbl_ext_recv_info, 47
  dbl_recv_info, 48

unit_protocol_info, 49