# DBL™
## Application Programming Interface

Version 3.1.6.52817

October 4, 2019

All information contained in this document is proprietary to CSP, Inc. and may not be reproduced, distributed, or disseminated, in whole or in part, without the written permission of an authorized representative of CSP, Inc.
All specifications presented in this document are subject to change at any time, and without prior notice.
Myricom® and Myrinet® are registered trademarks of CSP, Inc. DBL™ is a trademark of CSP, Inc. Other trademarks appearing in this document are those of their respective owners.
©2008-2014, CSP, Inc.

# Contents

# 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_gacdc677ef6b2d20f994ad45ca28373768}{dbl\_open}(interface, flags, &dev);
\hyperlink{group__DBL_gaaccc222ec7efc1dc2ed62f599ce3f0d7}{dbl\_bind}(dev, port1, flags, context1, &chan1);
\hyperlink{group__DBL_gaaccc222ec7efc1dc2ed62f599ce3f0d7}{dbl\_bind}(dev, port2, flags, context2, &chan2);
\textcolor{keywordflow}{while} (!done) \{
  \hyperlink{group__DBL_ga7c8fd37a2ca1147707688cb8b6a95bce}{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();

```
\hyperlink{group__DBL_gab9aed304b284dec7143ff83809a2d6fc}{dbl\_init}();
\hyperlink{group__DBL_gacdc677ef6b2d20f994ad45ca28373768}{dbl\_open}(interface, flags, &dev);
\hyperlink{group__DBL_gaaccc222ec7efc1dc2ed62f599ce3f0d7}{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_gab9aed304b284dec7143ff83809a2d6fc}{dbl\_init}();
\hyperlink{group__DBL_gacdc677ef6b2d20f994ad45ca28373768}{dbl\_open}(interface, flags, &dev);
\hyperlink{group__DBL_gaaccc222ec7efc1dc2ed62f599ce3f0d7}{dbl\_bind}(dev, port1, flags, context1, &chan1);
\hyperlink{group__DBL_gab1df9a3b4bc9a1fbb2e8a8f166f6cc31}{dbl\_send\_connect}(chan1, address, flags, ttl, &send\_
\hyperlink{group__DBL_gaf169475824a50f2663f5b6f82e084c06}{dbl\_send}(send\_handle, buf, buflen, flags);
```

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

```
\hyperlink{group__DBL_gab9aed304b284dec7143ff83809a2d6fc}{dbl\_init}();
\hyperlink{group__DBL_gacdc677ef6b2d20f994ad45ca28373768}{dbl\_open}(interface, flags, &dev);
\hyperlink{group__DBL_gaaccc222ec7efc1dc2ed62f599ce3f0d7}{dbl\_bind}(dev, port1, flags, context1, &chan1);
\hyperlink{group__DBL_gadfd63607d172bcdd1380904b2b673244}{dbl\_mcast\_join}(chan1, mcast\_addr, NULL);
\hyperlink{group__DBL_ga7c8fd37a2ca1147707688cb8b6a95bce}{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_gab9aed304b284dec7143ff83809a2d6fc}{dbl\_init}();
\hyperlink{group__DBL_gacdc677ef6b2d20f994ad45ca28373768}{dbl\_open}(interface, flags, &dev);
\hyperlink{group__DBL_gaaccc222ec7efc1dc2ed62f599ce3f0d7}{dbl\_bind}(dev, port1, flags, context1, &chan1);
\hyperlink{group__DBL_gaaccc222ec7efc1dc2ed62f599ce3f0d7}{dbl\_bind}(dev, port2, flags, context2, &chan2);
\hyperlink{group__DBL_gadfd63607d172bcdd1380904b2b673244}{dbl\_mcast\_join}(chan1, mcast\_addr1, NULL);
\hyperlink{group__DBL_gadfd63607d172bcdd1380904b2b673244}{dbl\_mcast\_join}(chan1, mcast\_addr2, NULL);
\hyperlink{group__DBL_gadfd63607d172bcdd1380904b2b673244}{dbl\_mcast\_join}(chan2, mcast\_addr1, NULL);
\hyperlink{group__DBL_gadfd63607d172bcdd1380904b2b673244}{dbl\_mcast\_join}(chan2, mcast\_addr3, NULL);
\hyperlink{group__DBL_ga7c8fd37a2ca1147707688cb8b6a95bce}{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 appropritate 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 duplcate 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:

# Chapter 3

# Namespace Index

## 3.1 Namespace List

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

# Chapter 4

# Data Structure Index

## 4.1 Data Structures

Here are the data structures with brief descriptions:

# 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 dep, 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

#### 5.1.4.1 enum dbl_filter_mode

Filtering modes (advanced functionality).

**Remarks**

Selecting anything bu 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**

| | |
|---|---|
| *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() 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**

| | |
|---|---|
| *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**

| | |
|---|---|
| *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 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.

#### 5.1.5.8 dbl_getaddress ( dbl_channel_t *ch,* struct sockaddr_in * *sin* )

Returns the address to which a channel is bound.

Returns the address to which a channel is bound.

**Parameters**

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

**Return values**

| | |
|---|---|
| *0* | Sucess |
| *EINVAL* | Bad channel specified |

**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.

Returns the current NIC time.

**Parameters**

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

**Return values**

| | |
|---|---|
| *0* | Sucess |
| *EINVAL* | Bad dev specified |

**Remarks**

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

#### 5.1.5.10 dbl_init ( uint16_t *api_version* )

Initializes the dbl library.

Initializes the dbl library.

**Parameters**

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

**Return values**

| | |
|---|---|
| *0* | Success |
| *EINVAL* | Bad/incompatible version passed. |

**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**

| | |
|---|---|
| *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**

| | |
|---|---|
| *0* | Success |
| *EINVAL* | Argument error, such as address not multicast group. |
| *EADDRNOTAVAIL* | Not currently joined to group "address" |
| *EAGAIN* | internal resources temorarily unavailable, try again. |
| *?* | Other non-zero codes indicate various OS failures in the leave process |

#### 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**

| | |
|---|---|
| *ch* | Handle for the channel to add to the specified multicast group. |
| *mcast_addr* | Address of the multicast group to join. |
| *unused* | A temporary unused pointer to maintain binary compability. |

**Return values**

| | |
|---|---|
| *0* | Success |
| *EINVAL* | Argument error, such as address is not a multicast group. |
| *?* | Other values indicate various OS specific failures in the join process. |

#### 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**

| | |
|---|---|
| *ch* | Handle for the channel to add to the specified multicast group. |
| *mcast_addr* | Address of the multicast group to join. |
| *src* | Address of source to receive multicast from |

**Return values**

| | |
|---|---|
| *0* | Success |
| *EINVAL* | Argument error, such as address is not a multicast group. |
| *?* | Other values indicate various OS specific failures in the join process. |

#### 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**

| | |
|---|---|
| *ch* | Handle for the channel to leave the specified multicast group. |
| *mcast_addr* | Address of the multicast group to leave. |

**Return values**

| | |
|---|---|
| *0* | Success |
| *EINVAL* | Argument error, such as address not multicast group. |
| *EADDRNOTAVAIL* | Not currently joined to group "address" |
| *EAGAIN* | internal resources temorarily unavailable, try again. |
| *?* | Other non-zero codes indicate various OS failures in the leave process |

#### 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**

| | |
|---|---|
| *ch* | Handle for the channel to leave the specified multicast group. |
| *mcast_addr* | Address of the multicast group to leave. |
| *src* | Address of the source to drop |

**Return values**

| | |
|---|---|
| *0* | Success |
| *EINVAL* | Argument error, such as address not multicast group. |
| *EADDRNOTAVAIL* | Not currently joined to group "address" |
| *EAGAIN* | internal resources temporarily unavailable, try again. |
| *?* | Other non-zero codes indicate various OS failures in the leave process |

#### 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**

| | |
|---|---|
| *ch* | Handle for the channel to leave the specified multicast group. |
| *join_addr* | Address of the multicast group to join. |
| *block_addr* | Address to unblock. The multicast packets will again be received from the unblocked source |

**Return values**

| | |
|---|---|
| *0* | Success |
| *EINVAL* | Argument error, such as address not multicast group. |
| *EADDRNOTAVAIL* | Not currently joined to group "address" |
| *EAGAIN* | internal resources temorarily unavailable, try again. |
| *?* | Other non-zero codes indicate various OS failures in the leave process |

#### 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**

| | |
|---|---|
| *0* | Success |
| *EINVAL* | bad usage. includes dbl_init not called first and bad interface_addr. |
| *ENODEV* | no matching IP address found on DBL-enabled NIC |
| *EAGAIN* | internal resources temorarily unavailable, try again. |

**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**

| | |
|---|---|
| *0* | Success |
| *EINVAL* | bad usage. includes dbl_init not called first and bad interface_addr. |
| *EAGAIN* | internal resources temorarily unavailable, try again. |

### Remarks

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

#### 5.1.5.19 dbl_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

| | |
|---|---|
| *dev* | The underlying device via dbl_open |
| *mode* | See dbl_recvmode |
| *buf* | Buffer in which to place received data. |
| *len* | Maximum number of bytes to write into buf. |
| *info* | See dbl_recv_info. |

### Return values

| | |
|---|---|
| *0* | Success |
| *EAGAIN* | Returned if using mode DBL_RECV_NONBLOCK or DBL_RECV_PEEK when no packet is available. |
| *EINTR* | in case dbl_shutdown() was called |
| *?* | Other codes indicate various OS failures. |

### 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 speficied 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 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

| | |
|---|---|
| *sendh* | Send handle specifying destination for packe. |
| *buf* | The data to send. |
| *len* | The number of bytes to send. |
| *flags* | See Flags for dbl_send().. |

**Return values**

| | |
|---|---|
| *0* | Success |
| *EAGAIN* | DBL_NONBLOCK specified and no resources available. |
| *?* | Other codes indicate various OS failures in the send process. |

**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**

| | |
|---|---|
| *0* | Sucess |
| *EINVAL* | Errors in arguments |
| *?* | Other codes indicate various OS failures in the send process. |

**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**

| | |
|---|---|
| *EISCONN* | channel already connected |

#### 5.1.5.22 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**

| | |
|---|---|
| *0* | Success |

**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_in * *sin*, const void * *buf*, size_t *len*, int *flags* )

Send a packet.

Send a packet to the address specified.

**Parameters**

| | |
|---|---|
| *ch* | Handle for the channel to send over. |
| *sin* | The destination address |
| *buf* | The data to send. |
| *len* | The lenght of the data to send. |
| *flags* | See Flags for dbl_send(). |

**Return values**

| | |
|---|---|
| *0* | Success |
| *EAGAIN* | DBL_NONBLOCK specified and no resources available. |
| *?* | Other codes indicate various OS failures in the send process. |

#### 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**

| | |
|---|---|
| *dev* | The underlying device via dbl_open |
| *how* | Unused for now |

**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**

| | |
|---|---|
| *handle* | The handle of the channel to unbind. |

**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

### 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 timestmaping 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_listen (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**

| | |
|---|---|
| *ch* | The channel (from dbl_bind()) on which connections are accepted |
| *sad* | 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. |
| *len* | 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. |
| *rcontext* | The value of rcontext is associated with the new channel |
| *rch* | The channel which can be used to communicate with the remote peer. |

**Return values**

| | |
|---|---|
| *0* | Success |
| *?* | Other codes indicate various OS failures. |

#### 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**

| | |
|---|---|
| *ch* | A valid channel |

**Return values**

| | |
|---|---|
| *1* | Channel is TCP |
| *0* | Otherwise |

#### 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**

| | |
|---|---|
| *ch* | The channel |

| | |
|---|---|
| *level* | Level of the option (IPPROTO_IP...) |
| *optname* | Option's name (IP_TTL...) |
| *optval* | The pointer on the value |
| *optlen* | The pointer on the option's length |

**Return values**

| | |
|---|---|
| == | 0 Success |
| > | 0 OS return code |

**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_ext_listen ( 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**

| | |
|---|---|
| *0* | Success |
| *?* | Other codes indicate various OS failures. |

#### 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**

| | |
|---|---|
| *ch* | The channel (from dbl_bind()) on which a packet has been received. |
| *mode* | See dbl_recvmode |
| *buf* | Buffer in which to place received data. |
| *len* | Maximum number of bytes to write into buf. |
| *info* | See dbl_recv_info. |

**Return values**

| | |
|---|---|
| *0* | Success |
| *EAGAIN* | Returned if using mode DBL_RECV_NONBLOCK or DBL_RECV_PEEK when no packet is available. |
| *?* | Other codes indicate various OS failures. |

**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 a array of receive information

**Parameters**

| | |
|---|---|
| *dev* | The device |
| *recv_mode* | See dbl_recvmode |
| *info* | 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, decribing then len of the buffer in input, and returning the len of the message copied (see dbl_recv_info) |
| *recvmax* | the number of message which can be loaded |

**Return values**

| | |
|---|---|
| >= | 0 number of messages to retrieve in the info array |
| < | 0 error should be retrieved in errno |

**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**

| | |
|---|---|
| *ch* | The connected channel |
| *buf* | pointer to buffer |
| *paylen* | size to send See Flags for dbl_send(). return the number of bytes sent |

**Return values**

| | |
|---|---|
| *0* | Success |
| *?* | Other codes indicate various OS failures. |

#### 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**

| | |
|---|---|
| *ch* | The channel |
| *level* | Level of the option (IPPROTO_IP...) |
| *optname* | Option's name (IP_TTL...) |
| *optval* | The pointer on the value |
| *optlen* | The option's type length |

**Return values**

| | |
|---|---|
| == | 0 Success |
| > | 0 OS return code |

**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.

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 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