NAME

BIO_s_bio, BIO_make_bio_pair, BIO_destroy_bio_pair, BIO_shutdown_wr, BIO_set_write_buf_size, BIO_get_write_buf_size, BIO_new_bio_pair, BIO_get_write_guarantee, BIO_ctrl_get_write_guarantee, BIO_get_read_request, BIO_ctrl_get_read_request, BIO_ctrl_reset_read_request, BIO_nread0, BIO_nread, BIO_nwrite0, BIO_nwrite − BIO pair BIO

SYNOPSIS

```
#include <openssl/bio.h>

const BIO_METHOD *BIO_s_bio(void);

int BIO_make_bio_pair(BIO *b1, BIO *b2);
int BIO_destroy_bio_pair(BIO *b);
int BIO_shutdown_wr(BIO *b);

int BIO_set_write_buf_size(BIO *b, long size);
size_t BIO_get_write_buf_size(BIO *b, long size);

int BIO_new_bio_pair(BIO **bio1, size_t writebuf1, BIO **bio2, size_t writebuf2);

int BIO_get_write_guarantee(BIO *b);
size_t BIO_ctrl_get_write_guarantee(BIO *b);
int BIO_get_read_request(BIO *b);
size_t BIO_ctrl_get_read_request(BIO *b);
int BIO_ctrl_reset_read_request(BIO *b);

int BIO_nread0(BIO *bio, char **buf);
int BIO_nread(BIO *bio, char **buf, int num);
int BIO_nwrite0(BIO *bio, char **buf);
int BIO_nwrite(BIO *bio, char **buf, int num);
```

DESCRIPTION

BIO_s_bio() returns the method for a BIO pair. A BIO pair is a pair of source/sink BIOs where data written to either half of the pair is buffered and can be read from the other half. Both halves must usually by handled by the same application thread since no locking is done on the internal data structures.

Since BIO chains typically end in a source/sink BIO it is possible to make this one half of a BIO pair and have all the data processed by the chain under application control.

One typical use of BIO pairs is to place TLS/SSL I/O under application control, this can be used when the application wishes to use a non standard transport for TLS/SSL or the normal socket routines are inappropriate.

Calls to **BIO_read_ex()** will read data from the buffer or request a retry if no data is available.

Calls to **BIO_write_ex()** will place data in the buffer or request a retry if the buffer is full.

The standard calls **BIO_ctrl_pending()** and **BIO_ctrl_wpending()** can be used to determine the amount of pending data in the read or write buffer.

BIO_reset() clears any data in the write buffer.

BIO_make_bio_pair() joins two separate BIOs into a connected pair.

BIO_destroy_pair() destroys the association between two connected BIOs. Freeing up any half of the pair will automatically destroy the association.

BIO_shutdown_wr() is used to close down a BIO **b**. After this call no further writes on BIO **b** are allowed (they will return an error). Reads on the other half of the pair will return any pending data or EOF when all pending data has been read.

BIO_set_write_buf_size() sets the write buffer size of BIO **b** to **size**. If the size is not initialized a default

value is used. This is currently 17K, sufficient for a maximum size TLS record.

BIO_get_write_buf_size() returns the size of the write buffer.

BIO_new_bio_pair() combines the calls to **BIO_new()**, **BIO_make_bio_pair()** and **BIO_set_write_buf_size()** to create a connected pair of BIOs **bio1**, **bio2** with write buffer sizes **writebuf1** and **writebuf2**. If either size is zero then the default size is used. **BIO_new_bio_pair()** does not check whether **bio1** or **bio2** do point to some other BIO, the values are overwritten, **BIO_free()** is not called.

BIO_get_write_guarantee() and **BIO_ctrl_get_write_guarantee()** return the maximum length of data that can be currently written to the BIO. Writes larger than this value will return a value from **BIO_write_ex()** less than the amount requested or if the buffer is full request a retry. **BIO_ctrl_get_write_guarantee()** is a function whereas **BIO_get_write_guarantee()** is a macro.

BIO_get_read_request() and **BIO_ctrl_get_read_request()** return the amount of data requested, or the buffer size if it is less, if the last read attempt at the other half of the BIO pair failed due to an empty buffer. This can be used to determine how much data should be written to the BIO so the next read will succeed: this is most useful in TLS/SSL applications where the amount of data read is usually meaningful rather than just a buffer size. After a successful read this call will return zero. It also will return zero once new data has been written satisfying the read request or part of it. Note that **BIO_get_read_request()** never returns an amount larger than that returned by **BIO_get_write_guarantee()**.

BIO_ctrl_reset_read_request() can also be used to reset the value returned by **BIO_get_read_request()** to zero.

Non−copying Interface

BIO_nread0(), **BIO_nread()**, **BIO_nwrite0()**, and **BIO_nwrite()** provide a non−copying interface for reading from and writing to BIO pairs. These functions allow direct access to the internal buffer, avoiding the overhead of copying data.

BIO_nread0() returns in ***buf** a pointer to the start of the available data in the peer's write buffer and returns the number of bytes available. This allows reading directly from the buffer without copying. It does not consume the data; a subsequent call to **BIO_nread()** is needed to advance the buffer position.

BIO_nread() is similar to **BIO_nread0()** but also advances the read position by up to **num** bytes. The actual number of bytes consumed is returned. The ***buf** pointer is set to the start of the data that was consumed. Since the data is considered consumed after this call, the pointer returned by**BIO_nr ead()** should not be used afterwards unless the caller also controls the writing side. The typical pattern is to call **BIO_nread0()** first, use the data, and then call **BIO_nread()** to consume it.

BIO_nwrite0() returns in ***buf** a pointer to the start of the available space in the write buffer and returns the number of bytes that can be written. This allows writing directly to the buffer without copying. It does not commit the data; a subsequent call to **BIO_nwrite()** is needed to update the buffer length.

BIO_nwrite() is similar to **BIO_nwrite0()** but also commits up to **num** bytes as written. The actual number of bytes committed is returned. The ***buf** pointer is set to the start of the region that was committed. **BIO_nwrite()** should only be called after the data has actually been written to the buffer obtained from **BIO_nwrite0()**, since committing signals data availability to the reading side.

Note that due to the ring buffer implementation, if wrapping around would be required, **BIO_nread0()** and **BIO_nwrite0()** may return less than the total available space. In such cases, a second call may be needed to access the remaining data or space.

NOTES

Both halves of a BIO pair should be freed. That is even if one half is implicit freed due to a **BIO_free_all()** or **SSL_free()** call the other half needs to be freed.

When used in bidirectional applications (such as TLS/SSL) care should be taken to flush any data in the write buffer. This can be done by calling **BIO_pending()** on the other half of the pair and, if any data is pending, reading it and sending it to the underlying transport. This must be done before any normal processing (such as calling **select()**) due to a request and **BIO_should_read()** being true.

To see why this is important consider a case where a request is sent using **BIO_write_ex()** and a response

read with **BIO_read_ex()**, this can occur during an TLS/SSL handshake for example. **BIO_write_ex()** will succeed and place data in the write buffer. **BIO_read_ex()** will initially fail and **BIO_should_read()** will be true. If the application then waits for data to be available on the underlying transport before flushing the write buffer it will never succeed because the request was never sent!

BIO_eof() is true if no data is in the peer BIO and the peer BIO has been shutdown.

BIO_make_bio_pair(), **BIO_destroy_bio_pair()**, **BIO_shutdown_wr()**, **BIO_set_write_buf_size()**, **BIO_get_write_buf_size()**, **BIO_get_write_guarantee()**, and **BIO_get_read_request()** are implemented as macros.

RETURN VALUES

BIO_new_bio_pair() returns 1 on success, with the new BIOs available in **bio1** and **bio2**, or 0 on failure, with NULL pointers stored into the locations for **bio1** and **bio2**. Check the error stack for more information.

[XXXXX: More return values need to be added here]

BIO_nread0() returns the number of bytes available for reading, 0 if the peer has closed and no data remains (EOF), or −1 if no data is currently available (retry may be appropriate). If the BIO is not initialized, −2 is returned.

BIO_nwrite0() returns the number of bytes of space available for writing, or −1 if no space is currently available (retry may be appropriate) or the BIO has been closed. If the BIO is not initialized, −2 is returned.

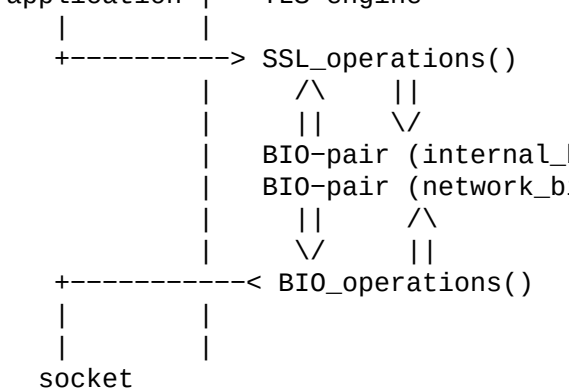
BIO_nread() and **BIO_nwrite()** return the number of bytes consumed or committed respectively, or the same error values as **BIO_nread0()** and **BIO_nwrite0()**.

EXAMPLES

The BIO pair can be used to have full control over the network access of an application. The application can call **select()** on the socket as required without having to go through the SSL−interface.

```
BIO *internal_bio, *network_bio;

...
BIO_new_bio_pair(&internal_bio, 0, &network_bio, 0);
SSL_set_bio(ssl, internal_bio, internal_bio);
SSL_operations(); /* e.g. SSL_read and SSL_write */
...
```




```
...
SSL_free(ssl);                /* implicitly frees internal_bio */
BIO_free(network_bio);
...
```

As the BIO pair will only buffer the data and never directly access the connection, it behaves nonblocking

and will return as soon as the write buffer is full or the read buffer is drained. Then the application has to flush the write buffer and/or fill the read buffer.

Use the **BIO_ctrl_pending()**, to find out whether data is buffered in the BIO and must be transferred to the network. Use **BIO_ctrl_get_read_request()** to find out, how many bytes must be written into the buffer before the **SSL_operation()** can successfully be continued.

A typical usage pattern for the non−copying write interface is:

```
int ret;
char *buf;

ret = BIO_nwrite0(bio, &buf);
if (ret > 0) {
    /* write up to 'ret' bytes directly to 'buf' */
    memcpy(buf, data, len);
    BIO_nwrite(bio, &buf, len);  /* commit the write */
}
```

A typical usage pattern for the non−copying read interface is:

```
int ret;
char *buf;

ret = BIO_nread0(bio, &buf);
if (ret > 0) {
    /* read up to 'ret' bytes directly from 'buf' */
    process_data(buf, ret);
    BIO_nread(bio, &buf, ret);  /* consume the data */
}
```

WARNINGS

As the data is buffered, **SSL_operation()** may return with an ERROR_SSL_WANT_READ condition, but there is still data in the write buffer. An application must not rely on the error value of **SSL_operation()** but must assure that the write buffer is always flushed first. Otherwise a deadlock may occur as the peer might be waiting for the data before being able to continue.

SEE ALSO

SSL_set_bio (3), **ssl** (7), **bio** (7), **BIO_should_retry** (3), **BIO_read_ex** (3)

COPYRIGHT