Fast portable non-blocking network programming with Libevent

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

```
_____________________________________
These documents are Copyright (c) 2009-2012 by Nick Mathewson, and are made
available under the Creative Commons Attribution-Noncommercial-Share Alike
license, version 3.0.  Future versions may be made available under a less
restrictive license.

Additionally, the source code examples in these documents are also licensed
under the so-called "3-Clause" or "Modified" BSD license. See
link:license_bsd.html[the license_bsd file] distributed with these documents
for the full terms.

For the latest version of this document, see
   http://www.wangafu.net/~nickm/libevent-book/TOC.html

To get the source for the latest version of this document, install git
and run "git clone git://github.com/nmathewson/libevent-book.git"
_____________________________________
```

```
This license applies to the example source code in the book,
   "Fast Portable non-blocking network programming in Libevent."
The book itself is NOT under this license.

Copyright 2009-2012 Nick Mathewson

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following disclaimer
in the documentation and/or other materials provided with the
distribution.

    * Neither the names of the copyright owners nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

To get the source for the latest version of this document, install git and run "git clone git://github.com/nmathewson/libevent-book.git"

About this document

This document will teach you how to use Libevent 2.0 (and later) to write fast portable asynchronous network IO programs in C. We assume:

- That you already know C.
- That you already know the basic C networking calls (socket(), connect(), and so on).

A note on examples

The examples in this document should work all right on Linux, FreeBSD, OpenBSD, NetBSD, Mac OS X, Solaris, and Android. Some of the examples may not compile on Windows.

A tiny introduction to asynchronous IO

Most beginning programmers start with blocking IO calls. An IO call is *synchronous* if, when you call it, it does not return until the operation is completed, or until enough time has passed that your network stack gives up. When you call "connect()" on a TCP connection, for example, your operating system queues a SYN packet to the host on the other side of the TCP connection. It does not return control back to your application until either it has received a SYN ACK packet from the opposite host, or until enough time has passed that it decides to give up.

Here's an example of a really simple client using blocking network calls. It opens a connection to www.google.com, sends it a simple HTTP request, and prints the response to stdout.

Example: A simple blocking HTTP client

```
/* For sockaddr_in */
#include <netinet/in.h>
/* For socket functions */
#include <sys/socket.h>
/* For gethostbyname */
#include <netdb.h>

#include <unistd.h>
#include <string.h>
#include <stdio.h>

int main(int c, char **v)
{
    const char query[] =
        "GET / HTTP/1.0\r\n"
```

```
        "Host: www.google.com\r\n"
        "\r\n";
    const char hostname[] = "www.google.com";
    struct sockaddr_in sin;
    struct hostent *h;
    const char *cp;
    int fd;
    ssize_t n_written, remaining;
    char buf[1024];

    /* Look up the IP address for the hostname.   Watch out; this isn't
       threadsafe on most platforms. */
    h = gethostbyname(hostname);
    if (!h) {
        fprintf(stderr, "Couldn't lookup %s: %s", hostname, hstrerror(h_errno));
        return 1;
    }
    if (h->h_addrtype != AF_INET) {
        fprintf(stderr, "No ipv6 support, sorry.");
        return 1;
    }

    /* Allocate a new socket */
    fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) {
        perror("socket");
        return 1;
    }

    /* Connect to the remote host. */
    sin.sin_family = AF_INET;
    sin.sin_port = htons(80);
    sin.sin_addr = *(struct in_addr*)h->h_addr;
    if (connect(fd, (struct sockaddr*) &sin, sizeof(sin))) {
        perror("connect");
        close(fd);
        return 1;
    }

    /* Write the query. */
    /* XXX Can send succeed partially? */
    cp = query;
    remaining = strlen(query);
    while (remaining) {
      n_written = send(fd, cp, remaining, 0);
      if (n_written <= 0) {
        perror("send");
        return 1;
      }
      remaining -= n_written;
      cp += n_written;
    }

    /* Get an answer back. */
    while (1) {
        ssize_t result = recv(fd, buf, sizeof(buf), 0);
        if (result == 0) {
            break;
        } else if (result < 0) {
            perror("recv");
            close(fd);
            return 1;
```

```
        }
        fwrite(buf, 1, result, stdout);
    }

    close(fd);
    return 0;
}
```

All of the network calls in the code above are *blocking*: the gethostbyname does not return until it has succeeded or failed in resolving www.google.com; the connect does not return until it has connected; the recv calls do not return until they have received data or a close; and the send call does not return until it has at least flushed its output to the kernel's write buffers.

Now, blocking IO is not necessarily evil. If there's nothing else you wanted your program to do in the meantime, blocking IO will work fine for you. But suppose that you need to write a program to handle multiple connections at once. To make our example concrete: suppose that you want to read input from two connections, and you don't know which connection will get input first. You can't say

Bad Example

```
/* This won't work. */
char buf[1024];
int i, n;
while (i_still_want_to_read()) {
    for (i=0; i<n_sockets; ++i) {
        n = recv(fd[i], buf, sizeof(buf), 0);
        if (n==0)
            handle_close(fd[i]);
        else if (n<0)
            handle_error(fd[i], errno);
        else
            handle_input(fd[i], buf, n);
    }
}
```

because if data arrives on fd[2] first, your program won't even try reading from fd[2] until the reads from fd[0] and fd[1] have gotten some data and finished.

Sometimes people solve this problem with multithreading, or with multi-process servers. One of the simplest ways to do multithreading is with a separate process (or thread) to deal with each connection. Since each connection has its own process, a blocking IO call that waits for one connection won't make any of the other connections' processes block.

Here's another example program. It is a trivial server that listens for TCP connections on port 40713, reads data from its input one line at a time, and writes out the ROT13 obfuscation of line each as it arrives. It uses the Unix fork() call to create a new process for each incoming connection.

Example: Forking ROT13 server

```
/* For sockaddr_in */
#include <netinet/in.h>
/* For socket functions */
#include <sys/socket.h>

#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_LINE 16384

char
rot13_char(char c)
{
    /* We don't want to use isalpha here; setting the locale would change
```

```
     * which characters are considered alphabetical. */
    if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
        return c + 13;
    else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
        return c - 13;
    else
        return c;
}

void
child(int fd)
{
    char outbuf[MAX_LINE+1];
    size_t outbuf_used = 0;
    ssize_t result;

    while (1) {
        char ch;
        result = recv(fd, &ch, 1, 0);
        if (result == 0) {
            break;
        } else if (result == -1) {
            perror("read");
            break;
        }

        /* We do this test to keep the user from overflowing the buffer. */
        if (outbuf_used < sizeof(outbuf)) {
            outbuf[outbuf_used++] = rot13_char(ch);
        }

        if (ch == '\n') {
            send(fd, outbuf, outbuf_used, 0);
            outbuf_used = 0;
            continue;
        }
    }
}

void
run(void)
{
    int listener;
    struct sockaddr_in sin;

    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = 0;
    sin.sin_port = htons(40713);

    listener = socket(AF_INET, SOCK_STREAM, 0);

#ifndef WIN32
    {
        int one = 1;
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
    }
#endif

    if (bind(listener, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
        perror("bind");
        return;
    }
```

```
    if (listen(listener, 16)<0) {
        perror("listen");
        return;
    }



    while (1) {
        struct sockaddr_storage ss;
        socklen_t slen = sizeof(ss);
        int fd = accept(listener, (struct sockaddr*)&ss, &slen);
        if (fd < 0) {
            perror("accept");
        } else {
            if (fork() == 0) {
                child(fd);
                exit(0);
            }
        }
    }
}

int
main(int c, char **v)
{
    run();
    return 0;
}
```

So, do we have the perfect solution for handling multiple connections at once? Can I stop writing this book and go work on something else now? Not quite. First off, process creation (and even thread creation) can be pretty expensive on some platforms. In real life, you'd want to use a thread pool instead of creating new processes. But more fundamentally, threads won't scale as much as you'd like. If your program needs to handle thousands or tens of thousands of connections at a time, dealing with tens of thousands of threads will not be as efficient as trying to have only a few threads per CPU.

But if threading isn't the answer to having multiple connections, what is? In the Unix paradigm, you make your sockets *non-blocking*. The Unix call to do this is:

```
fcntl(fd, F_SETFL, O_NONBLOCK);
```

where fd is the file descriptor for the socket. [1] Once you've made fd (the socket) nonblocking, from then on, whenever you make a network call to fd the call will either complete the operation immediately or return with a special error code to indicate "I couldn't make any progress now, try again." So our two-socket example might be naively written as:

Bad Example: busy-polling all sockets

```
/* This will work, but the performance will be unforgivably bad. */
int i, n;
char buf[1024];
for (i=0; i < n_sockets; ++i)
    fcntl(fd[i], F_SETFL, O_NONBLOCK);

while (i_still_want_to_read()) {
    for (i=0; i < n_sockets; ++i) {
        n = recv(fd[i], buf, sizeof(buf), 0);
        if (n == 0) {
            handle_close(fd[i]);
        } else if (n < 0) {
            if (errno == EAGAIN)
```

[1] A file descriptor is the number the kernel assigns to the socket when you open it. You use this number to make Unix calls referring to the socket.

```
                ; /* The kernel didn't have any data for us to read. */
            else
                handle_error(fd[i], errno);
        } else {
            handle_input(fd[i], buf, n);
        }
    }
}
```

Now that we're using nonblocking sockets, the code above would *work*... but only barely. The performance will be awful, for two reasons. First, when there is no data to read on either connection the loop will spin indefinitely, using up all your CPU cycles. Second, if you try to handle more than one or two connections with this approach you'll do a kernel call for each one, whether it has any data for you or not. So what we need is a way to tell the kernel "wait until one of these sockets is ready to give me some data, and tell me which ones are ready."

The oldest solution that people still use for this problem is select(). The select() call takes three sets of fds (implemented as bit arrays): one for reading, one for writing, and one for "exceptions". It waits until a socket from one of the sets is ready and alters the sets to contain only the sockets ready for use.

Here is our example again, using select:

Example: Using select

```
/* If you only have a couple dozen fds, this version won't be awful */
fd_set readset;
int i, n;
char buf[1024];

while (i_still_want_to_read()) {
    int maxfd = -1;
    FD_ZERO(&readset);

    /* Add all of the interesting fds to readset */
    for (i=0; i < n_sockets; ++i) {
         if (fd[i]>maxfd) maxfd = fd[i];
         FD_SET(fd[i], &readset);
    }

    /* Wait until one or more fds are ready to read */
    select(maxfd+1, &readset, NULL, NULL, NULL);

    /* Process all of the fds that are still set in readset */
    for (i=0; i < n_sockets; ++i) {
        if (FD_ISSET(fd[i], &readset)) {
            n = recv(fd[i], buf, sizeof(buf), 0);
            if (n == 0) {
                handle_close(fd[i]);
            } else if (n < 0) {
                if (errno == EAGAIN)
                     ; /* The kernel didn't have any data for us to read. */
                else
                     handle_error(fd[i], errno);
             } else {
                handle_input(fd[i], buf, n);
             }
        }
    }
}
```

And here's a reimplementation of our ROT13 server, using select() this time.

Example: select()-based ROT13 server

```
/* For sockaddr_in */
#include <netinet/in.h>
/* For socket functions */
#include <sys/socket.h>
/* For fcntl */
#include <fcntl.h>
/* for select */
#include <sys/select.h>

#include <assert.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#define MAX_LINE 16384

char
rot13_char(char c)
{
    /* We don't want to use isalpha here; setting the locale would change
     * which characters are considered alphabetical. */
    if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
        return c + 13;
    else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
        return c - 13;
    else
        return c;
}

struct fd_state {
    char buffer[MAX_LINE];
    size_t buffer_used;

    int writing;
    size_t n_written;
    size_t write_upto;
};

struct fd_state *
alloc_fd_state(void)
{
    struct fd_state *state = malloc(sizeof(struct fd_state));
    if (!state)
        return NULL;
    state->buffer_used = state->n_written = state->writing =
        state->write_upto = 0;
    return state;
}

void
free_fd_state(struct fd_state *state)
{
    free(state);
}

void
make_nonblocking(int fd)
{
    fcntl(fd, F_SETFL, O_NONBLOCK);
```

```
}

int
do_read(int fd, struct fd_state *state)
{
    char buf[1024];
    int i;
    ssize_t result;
    while (1) {
        result = recv(fd, buf, sizeof(buf), 0);
        if (result <= 0)
            break;

        for (i=0; i < result; ++i)  {
            if (state->buffer_used < sizeof(state->buffer))
                state->buffer[state->buffer_used++] = rot13_char(buf[i]);
            if (buf[i] == '\n') {
                state->writing = 1;
                state->write_upto = state->buffer_used;
            }
        }
    }

    if (result == 0) {
        return 1;
    } else if (result < 0) {
        if (errno == EAGAIN)
            return 0;
        return -1;
    }

    return 0;
}

int
do_write(int fd, struct fd_state *state)
{
    while (state->n_written < state->write_upto) {
        ssize_t result = send(fd, state->buffer + state->n_written,
                              state->write_upto - state->n_written, 0);
        if (result < 0) {
            if (errno == EAGAIN)
                return 0;
            return -1;
        }
        assert(result != 0);

        state->n_written += result;
    }

    if (state->n_written == state->buffer_used)
        state->n_written = state->write_upto = state->buffer_used = 0;

    state->writing = 0;

    return 0;
}

void
run(void)
{
    int listener;
```

```
    struct fd_state *state[FD_SETSIZE];
    struct sockaddr_in sin;
    int i, maxfd;
    fd_set readset, writeset, exset;

    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = 0;
    sin.sin_port = htons(40713);

    for (i = 0; i < FD_SETSIZE; ++i)
        state[i] = NULL;

    listener = socket(AF_INET, SOCK_STREAM, 0);
    make_nonblocking(listener);

#ifndef WIN32
    {
        int one = 1;
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
    }
#endif

    if (bind(listener, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
        perror("bind");
        return;
    }

    if (listen(listener, 16)<0) {
        perror("listen");
        return;
    }

    FD_ZERO(&readset);
    FD_ZERO(&writeset);
    FD_ZERO(&exset);

    while (1) {
        maxfd = listener;

        FD_ZERO(&readset);
        FD_ZERO(&writeset);
        FD_ZERO(&exset);

        FD_SET(listener, &readset);

        for (i=0; i < FD_SETSIZE; ++i) {
            if (state[i]) {
                if (i > maxfd)
                    maxfd = i;
                FD_SET(i, &readset);
                if (state[i]->writing) {
                    FD_SET(i, &writeset);
                }
            }
        }

        if (select(maxfd+1, &readset, &writeset, &exset, NULL) < 0) {
            perror("select");
            return;
        }

        if (FD_ISSET(listener, &readset)) {
```

```
            struct sockaddr_storage ss;
            socklen_t slen = sizeof(ss);
            int fd = accept(listener, (struct sockaddr*)&ss, &slen);
            if (fd < 0) {
                perror("accept");
            } else if (fd > FD_SETSIZE) {
                close(fd);
            } else {
                make_nonblocking(fd);
                state[fd] = alloc_fd_state();
                assert(state[fd]);/*XXX*/
            }
        }

        for (i=0; i < maxfd+1; ++i) {
            int r = 0;
            if (i == listener)
                continue;

            if (FD_ISSET(i, &readset)) {
                r = do_read(i, state[i]);
            }
            if (r == 0 && FD_ISSET(i, &writeset)) {
                r = do_write(i, state[i]);
            }
            if (r) {
                free_fd_state(state[i]);
                state[i] = NULL;
                close(i);
            }
        }
    }
}

int
main(int c, char **v)
{
    setvbuf(stdout, NULL, _IONBF, 0);

    run();
    return 0;
}
```

But we're still not done. Because generating and reading the select() bit arrays takes time proportional to the largest fd that you provided for select(), the select() call scales terribly when the number of sockets is high. [2]

Different operating systems have provided different replacement functions for select. These include poll(), epoll(), kqueue(), evports, and /dev/poll. All of these give better performance than select(), and all but poll() give O(1) performance for adding a socket, removing a socket, and for noticing that a socket is ready for IO.

Unfortunately, none of the efficient interfaces is a ubiquitous standard. Linux has epoll(), the BSDs (including Darwin) have kqueue(), Solaris has evports and /dev/poll... *and none of these operating systems has any of the others*. So if you want to write a portable high-performance asynchronous application, you'll need an abstraction that wraps all of these interfaces, and provides whichever one of them is the most efficient.

And that's what the lowest level of the Libevent API does for you. It provides a consistent interface to various select() replacements, using the most efficient version available on the computer where it's running.

Here's yet another version of our asynchronous ROT13 server. This time, it uses Libevent 2 instead of select(). Note that the

[2] On the userspace side, generating and reading the bit arrays can be made to take time proportional to the number of fds that you provided for select(). But on the kernel side, reading the bit arrays takes time proportional to the largest fd in the bit array, which tends to be around *the total number of fds in use in the whole program*, regardless of how many fds are added to the sets in select().

fd_sets are gone now: instead, we associate and disassociate events with a struct event_base, which might be implemented in terms of select(), poll(), epoll(), kqueue(), etc.

Example: A low-level ROT13 server with Libevent

```
/* For sockaddr_in */
#include <netinet/in.h>
/* For socket functions */
#include <sys/socket.h>
/* For fcntl */
#include <fcntl.h>

#include <event2/event.h>

#include <assert.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#define MAX_LINE 16384

void do_read(evutil_socket_t fd, short events, void *arg);
void do_write(evutil_socket_t fd, short events, void *arg);

char
rot13_char(char c)
{
    /* We don't want to use isalpha here; setting the locale would change
     * which characters are considered alphabetical. */
    if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
        return c + 13;
    else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
        return c - 13;
    else
        return c;
}

struct fd_state {
    char buffer[MAX_LINE];
    size_t buffer_used;

    size_t n_written;
    size_t write_upto;

    struct event *read_event;
    struct event *write_event;
};

struct fd_state *
alloc_fd_state(struct event_base *base, evutil_socket_t fd)
{
    struct fd_state *state = malloc(sizeof(struct fd_state));
    if (!state)
        return NULL;
    state->read_event = event_new(base, fd, EV_READ|EV_PERSIST, do_read, state);
    if (!state->read_event) {
        free(state);
        return NULL;
    }
    state->write_event =
        event_new(base, fd, EV_WRITE|EV_PERSIST, do_write, state);
```

```
    if (!state->write_event) {
        event_free(state->read_event);
        free(state);
        return NULL;
    }

    state->buffer_used = state->n_written = state->write_upto = 0;

    assert(state->write_event);
    return state;
}

void
free_fd_state(struct fd_state *state)
{
    event_free(state->read_event);
    event_free(state->write_event);
    free(state);
}

void
do_read(evutil_socket_t fd, short events, void *arg)
{
    struct fd_state *state = arg;
    char buf[1024];
    int i;
    ssize_t result;
    while (1) {
        assert(state->write_event);
        result = recv(fd, buf, sizeof(buf), 0);
        if (result <= 0)
            break;

        for (i=0; i < result; ++i)  {
            if (state->buffer_used < sizeof(state->buffer))
                state->buffer[state->buffer_used++] = rot13_char(buf[i]);
            if (buf[i] == '\n') {
                assert(state->write_event);
                event_add(state->write_event, NULL);
                state->write_upto = state->buffer_used;
            }
        }
    }

    if (result == 0) {
        free_fd_state(state);
    } else if (result < 0) {
        if (errno == EAGAIN) // XXXX use evutil macro
            return;
        perror("recv");
        free_fd_state(state);
    }
}

void
do_write(evutil_socket_t fd, short events, void *arg)
{
    struct fd_state *state = arg;

    while (state->n_written < state->write_upto) {
        ssize_t result = send(fd, state->buffer + state->n_written,
```

```
                                state->write_upto - state->n_written, 0);
        if (result < 0) {
            if (errno == EAGAIN) // XXX use evutil macro
                return;
            free_fd_state(state);
            return;
        }
        assert(result != 0);

        state->n_written += result;
    }

    if (state->n_written == state->buffer_used)
        state->n_written = state->write_upto = state->buffer_used = 1;

    event_del(state->write_event);
}

void
do_accept(evutil_socket_t listener, short event, void *arg)
{
    struct event_base *base = arg;
    struct sockaddr_storage ss;
    socklen_t slen = sizeof(ss);
    int fd = accept(listener, (struct sockaddr*)&ss, &slen);
    if (fd < 0) { // XXXX eagain??
        perror("accept");
    } else if (fd > FD_SETSIZE) {
        close(fd); // XXX replace all closes with EVUTIL_CLOSESOCKET */
    } else {
        struct fd_state *state;
        evutil_make_socket_nonblocking(fd);
        state = alloc_fd_state(base, fd);
        assert(state); /*XXX err*/
        assert(state->write_event);
        event_add(state->read_event, NULL);
    }
}

void
run(void)
{
    evutil_socket_t listener;
    struct sockaddr_in sin;
    struct event_base *base;
    struct event *listener_event;

    base = event_base_new();
    if (!base)
        return; /*XXXerr*/

    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = 0;
    sin.sin_port = htons(40713);

    listener = socket(AF_INET, SOCK_STREAM, 0);
    evutil_make_socket_nonblocking(listener);

#ifndef WIN32
    {
        int one = 1;
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
```

```
    }
#endif

    if (bind(listener, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
        perror("bind");
        return;
    }

    if (listen(listener, 16)<0) {
        perror("listen");
        return;
    }

    listener_event = event_new(base, listener, EV_READ|EV_PERSIST, do_accept, (void*)base);
    /*XXX check it */
    event_add(listener_event, NULL);

    event_base_dispatch(base);
}

int
main(int c, char **v)
{
    setvbuf(stdout, NULL, _IONBF, 0);

    run();
    return 0;
}
```

(Other things to note in the code: instead of typing the sockets as "int", we're using the type evutil_socket_t. Instead of calling fcntl(O_NONBLOCK) to make the sockets nonblocking, we're calling evutil_make_socket_nonblocking. These changes make our code compatible with the divergent parts of the Win32 networking API.)

What about convenience? (and what about Windows?)

You've probably noticed that as our code has gotten more efficient, it has also gotten more complex. Back when we were forking, we didn't have to manage a buffer for each connection: we just had a separate stack-allocated buffer for each process. We didn't need to explicitly track whether each socket was reading or writing: that was implicit in our location in the code. And we didn't need a structure to track how much of each operation had completed: we just used loops and stack variables.

Moreover, if you're deeply experienced with networking on Windows, you'll realize that Libevent probably isn't getting optimal performance when it's used as in the example above. On Windows, the way you do fast asynchronous IO is not with a select()-like interface: it's by using the IOCP (IO Completion Ports) API. Unlike all the fast networking APIs, IOCP does not alert your program when a socket is *ready* for an operation that your program then has to perform. Instead, the program tells the Windows networking stack to *start* a network operation, and IOCP tells the program when the operation has finished.

Fortunately, the Libevent 2 "bufferevents" interface solves both of these issues: it makes programs much simpler to write, and provides an interface that Libevent can implement efficiently on Windows *and* on Unix.

Here's our ROT13 server one last time, using the bufferevents API.

Example: A simpler ROT13 server with Libevent

```
/* For sockaddr_in */
#include <netinet/in.h>
/* For socket functions */
#include <sys/socket.h>
/* For fcntl */
#include <fcntl.h>

#include <event2/event.h>
```

```
#include <event2/buffer.h>
#include <event2/bufferevent.h>

#include <assert.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#define MAX_LINE 16384

void do_read(evutil_socket_t fd, short events, void *arg);
void do_write(evutil_socket_t fd, short events, void *arg);

char
rot13_char(char c)
{
    /* We don't want to use isalpha here; setting the locale would change
     * which characters are considered alphabetical. */
    if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
        return c + 13;
    else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
        return c - 13;
    else
        return c;
}

void
readcb(struct bufferevent *bev, void *ctx)
{
    struct evbuffer *input, *output;
    char *line;
    size_t n;
    int i;
    input = bufferevent_get_input(bev);
    output = bufferevent_get_output(bev);

    while ((line = evbuffer_readln(input, &n, EVBUFFER_EOL_LF))) {
        for (i = 0; i < n; ++i)
            line[i] = rot13_char(line[i]);
        evbuffer_add(output, line, n);
        evbuffer_add(output, "\n", 1);
        free(line);
    }

    if (evbuffer_get_length(input) >= MAX_LINE) {
        /* Too long; just process what there is and go on so that the buffer
         * doesn't grow infinitely long. */
        char buf[1024];
        while (evbuffer_get_length(input)) {
            int n = evbuffer_remove(input, buf, sizeof(buf));
            for (i = 0; i < n; ++i)
                buf[i] = rot13_char(buf[i]);
            evbuffer_add(output, buf, n);
        }
        evbuffer_add(output, "\n", 1);
    }
}

void
errorcb(struct bufferevent *bev, short error, void *ctx)
```

```
{
    if (error & BEV_EVENT_EOF) {
        /* connection has been closed, do any clean up here */
        /* ... */
    } else if (error & BEV_EVENT_ERROR) {
        /* check errno to see what error occurred */
        /* ... */
    } else if (error & BEV_EVENT_TIMEOUT) {
        /* must be a timeout event handle, handle it */
        /* ... */
    }
    bufferevent_free(bev);
}

void
do_accept(evutil_socket_t listener, short event, void *arg)
{
    struct event_base *base = arg;
    struct sockaddr_storage ss;
    socklen_t slen = sizeof(ss);
    int fd = accept(listener, (struct sockaddr*)&ss, &slen);
    if (fd < 0) {
        perror("accept");
    } else if (fd > FD_SETSIZE) {
        close(fd);
    } else {
        struct bufferevent *bev;
        evutil_make_socket_nonblocking(fd);
        bev = bufferevent_socket_new(base, fd, BEV_OPT_CLOSE_ON_FREE);
        bufferevent_setcb(bev, readcb, NULL, errorcb, NULL);
        bufferevent_setwatermark(bev, EV_READ, 0, MAX_LINE);
        bufferevent_enable(bev, EV_READ|EV_WRITE);
    }
}

void
run(void)
{
    evutil_socket_t listener;
    struct sockaddr_in sin;
    struct event_base *base;
    struct event *listener_event;

    base = event_base_new();
    if (!base)
        return; /*XXXerr*/

    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = 0;
    sin.sin_port = htons(40713);

    listener = socket(AF_INET, SOCK_STREAM, 0);
    evutil_make_socket_nonblocking(listener);

#ifndef WIN32
    {
        int one = 1;
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
    }
#endif

    if (bind(listener, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
```

```
        perror("bind");
        return;
    }

    if (listen(listener, 16)<0) {
        perror("listen");
        return;
    }

    listener_event = event_new(base, listener, EV_READ|EV_PERSIST, do_accept, (void*)base);
    /*XXX check it */
    event_add(listener_event, NULL);

    event_base_dispatch(base);
}

int
main(int c, char **v)
{
    setvbuf(stdout, NULL, _IONBF, 0);

    run();
    return 0;
}
```

How efficient is all of this, really?

XXXX write an efficiency section here. The benchmarks on the libevent page are really out of date.

The Libevent Reference Manual: Preliminaries

Libevent from 10,000 feet

Libevent is a library for writing fast portable nonblocking IO. Its design goals are:

Portability
: A program written using Libevent should work across all the platforms Libevent supports. Even when there is no really *good* way to do nonblocking IO, Libevent should support the so-so ways, so that your program can run in restricted environments.

Speed
: Libevent tries to use the fastest available nonblocking IO implementations on each platform, and not to introduce much overhead as it does so.

Scalability
: Libevent is designed to work well even with programs that need to have tens of thousands of active sockets.

Convenience
Whenever possible, the most natural way to write a program with Libevent should be the stable, portable way.

Libevent is divided into the following components:

evutil
Generic functionality to abstract out the differences between different platforms' networking implementations.

event and event_base
This is the heart of Libevent. It provides an abstract API to the various platform-specific, event-based nonblocking IO backends. It can let you know when sockets are ready to read or write, do basic timeout functionality, and detect OS signals.

bufferevent
These functions provide a more convenient wrapper around Libevent's event-based core. They let your application request buffered reads and writes, and rather than informing you when sockets are ready to do, they let you know when IO has actually occurred.

```
The bufferevent interface also has multiple backends, so that
it can take advantage of systems that provide faster ways to do
nonblocking IO, such as the Windows IOCP API.
```

evbuffer
This module implements the buffers underlying bufferevents, and provides functions for efficient and/or convenient access.

evhttp
A simple HTTP client/server implementation.

evdns
A simple DNS client/server implementation.

evrpc
A simple RPC implementation.

The Libraries

When Libevent is built, by default it installs the following libraries:

libevent_core
All core event and buffer functionality. This library contains all the event_base, evbuffer, bufferevent, and utility functions.

libevent_extra
This library defines protocol-specific functionality that you may or may not want for your application, including HTTP, DNS, and RPC.

libevent
This library exists for historical reasons; it contains the contents of both libevent_core and libevent_extra. You shouldn't use it; it may go away in a future version of Libevent.

The following libraries are installed only on some platforms:

libevent_pthreads
This library adds threading and locking implementations based on the pthreads portable threading library. It is separated from libevent_core so that you don't need to link against pthreads to use Libevent unless you are *actually* using Libevent in a multithreaded way.

libevent_openssl
This library provides support for encrypted communications using bufferevents and the OpenSSL library. It is separated from libevent_core so that you don't need to link against OpenSSL to use Libevent unless you are *actually* using encrypted connections.

The Headers

All current public Libevent headers are installed under the *event2* directory. Headers fall into three broad classes:

API headers
An API header is one that defines current public interfaces to Libevent. These headers have no special suffix.

Compatibility headers
A compatibility header includes definitions for deprecated functions. You shouldn't include it unless you're porting a program from an older version of Libevent.

Structure headers
These headers define structures with relatively volatile layouts. Some of these are exposed in case you need fast access to structure component; some are exposed for historical reasons. Relying on any of the structures in headers directly can break your program's binary compatibility with other versions of Libevent, sometimes in hard-to-debug ways. These headers have the suffix "_struct.h"

(There are also older versions of the Libevent headers without the *event2* directory. See "If you have to work with an old version of Libevent" below.)

If you have to work with an old version of Libevent

Libevent 2.0 has revised its APIs to be generally more rational and less error-prone. If it's possible, you should write new programs to use the Libevent 2.0 APIs. But sometimes you might need to work with the older APIs, either to update an existing application, or to support an environment where for some reason you can't install Libevent 2.0 or later.

Older versions of Libevent had fewer headers, and did not install them under "event2":

OLD HEADER…	…REPLACED BY CURRENT HEADERS
event.h	event2/event*.h, event2/buffer*.h event2/bufferevent*.h event2/tag*.h
evdns.h	event2/dns*.h
evhttp.h	event2/http*.h
evrpc.h	event2/rpc*.h
evutil.h	event2/util*.h

In Libevent 2.0 and later, the old headers still exist as wrappers for the new headers.

Some other notes on working with older versions:

- Before 1.4, there was only one library, "libevent", that contained the functionality currently split into libevent_core and libevent_extra.
- Before 2.0, there was no support for locking; Libevent could be thread-safe, but only if you made sure to never use the same structure from two threads at the same time.

Individual sections below will discuss the obsolete APIs that you might encounter for specific areas of the codebase.

Notes on version status

Versions of Libevent before 1.4.7 or so should be considered totally obsolete. Versions of Libevent before 1.3e or so should be considered hopelessly bug-ridden.

(Also, please don't send the Libevent maintainers any new features for 1.4.x or earlier---it's supposed to stay as a stable release. And if you encounter a bug in 1.3x or earlier, please make sure that it still exists in the latest stable version before you report it: subsequent releases have happened for a reason.)

Setting up the Libevent library

Libevent has a few global settings that are shared across the entire process. These affect the entire library.

You **must** make any changes to these settings before you call any other part of the Libevent library. If you don't, Libevent could wind up in an inconsistent state.

Log messages in Libevent

Libevent can log internal errors and warnings. It also logs debugging messages if it was compiled with logging support. By default, these messages are written to stderr. You can override this behavior by providing your own logging function.

Interface

```
#define EVENT_LOG_DEBUG 0
#define EVENT_LOG_MSG   1
#define EVENT_LOG_WARN  2
#define EVENT_LOG_ERR   3

/* Deprecated; see note at the end of this section */
#define _EVENT_LOG_DEBUG EVENT_LOG_DEBUG
#define _EVENT_LOG_MSG   EVENT_LOG_MSG
#define _EVENT_LOG_WARN  EVENT_LOG_WARN
#define _EVENT_LOG_ERR   EVENT_LOG_ERR

typedef void (*event_log_cb)(int severity, const char *msg);

void event_set_log_callback(event_log_cb cb);
```

To override Libevent's logging behavior, write your own function matching the signature of event_log_cb, and pass it as an argument to event_set_log_callback(). Whenever Libevent wants to log a message, it will pass it to the function you provided. You can have Libevent return to its default behavior by calling event_set_log_callback() again with NULL as an argument.

Examples

```
#include <event2/event.h>
#include <stdio.h>

static void discard_cb(int severity, const char *msg)
{
    /* This callback does nothing. */
}

static FILE *logfile = NULL;
static void write_to_file_cb(int severity, const char *msg)
{
    const char *s;
    if (!logfile)
        return;
```

```
    switch (severity) {
        case _EVENT_LOG_DEBUG: s = "debug"; break;
        case _EVENT_LOG_MSG:   s = "msg";   break;
        case _EVENT_LOG_WARN:  s = "warn";  break;
        case _EVENT_LOG_ERR:   s = "error"; break;
        default:               s = "?";     break; /* never reached */
    }
    fprintf(logfile, "[%s] %s\n", s, msg);
}

/* Turn off all logging from Libevent. */
void suppress_logging(void)
{
    event_set_log_callback(discard_cb);
}

/* Redirect all Libevent log messages to the C stdio file 'f'. */
void set_logfile(FILE *f)
{
    logfile = f;
    event_set_log_callback(write_to_file_cb);
}
```

NOTE It is not safe to invoke Libevent functions from within a user-provided event_log_cb callback! For instance, if you try to write a log callback that uses bufferevents to send warning messages to a network socket, you are likely to run into strange and hard-to-diagnose bugs. This restriction may be removed for some functions in a future version of Libevent.

Ordinarily, debug logs are not enabled, and are not sent to the logging callback. You can turn them on manually, if Libevent was built to support them.

Interface

```
#define EVENT_DBG_NONE 0
#define EVENT_DBG_ALL 0xffffffffu

void event_enable_debug_logging(ev_uint32_t which);
```

Debugging logs are verbose, and not necessarily useful under most circumstances. Calling event_enable_debug_logging() with EVENT_DBG_NONE gets default behavior; calling it with EVENT_DBG_ALL turns on all the supported debugging logs. More fine-grained options may be supported in future versions.

These functions are declared in <event2/event.h>. They first appeared in Libevent 1.0c, except for event_enable_debug_logging(), which first appeared in Libevent 2.1.1-alpha.

COMPATIBILITY NOTE Before Libevent 2.0.19-stable, the EVENT_LOG_* macros had names that began with an underscore: _EVENT_LOG_DEBUG, _EVENT_LOG_MSG, _EVENT_LOG_WARN, and _EVENT_LOG_ERR. These older names are deprecated, and should only be used for backward compatibility with Libevent 2.0.18-stable and earlier. They may be removed in a future version of Libevent.

Handling fatal errors

When Libevent detects a non-recoverable internal error (such as a corrupted data structure), its default behavior is to call exit() or abort() to leave the currently running process. These errors almost always mean that there is a bug somewhere: either in your code, or in Libevent itself.

You can override Libevent's behavior if you want your application to handle fatal errors more gracefully, by providing a function that Libevent should call in lieu of exiting.

Interface

```
typedef void (*event_fatal_cb)(int err);
void event_set_fatal_callback(event_fatal_cb cb);
```

To use these functions, you first define a new function that Libevent should call upon encountering a fatal error, then you pass it to event_set_fatal_callback(). Later, if Libevent encounters a fatal error, it will call the function you provided.

Your function **should not** return control to Libevent; doing so may cause undefined behavior, and Libevent might exit anyway to avoid crashing. Once your function has been called, you should not call any other Libevent function.

These functions are declared in <event2/event.h>. They first appeared in Libevent 2.0.3-alpha.

Memory management

By default, Libevent uses the C library's memory management functions to allocate memory from the heap. You can have Libevent use another memory manager by providing your own replacements for malloc, realloc, and free. You might want to do this if you have a more efficient allocator that you want Libevent to use, or if you have an instrumented allocator that you want Libevent to use in order to look for memory leaks.

Interface

```
void event_set_mem_functions(void *(*malloc_fn)(size_t sz),
                             void *(*realloc_fn)(void *ptr, size_t sz),
                             void (*free_fn)(void *ptr));
```

Here's a simple example that replaces Libevent's allocation functions with variants that count the total number of bytes that are allocated. In reality, you'd probably want to add locking here to prevent errors when Libevent is running in multiple threads.

Example

```
#include <event2/event.h>
#include <sys/types.h>
#include <stdlib.h>

/* This union's purpose is to be as big as the largest of all the
 * types it contains. */
union alignment {
    size_t sz;
    void *ptr;
    double dbl;
};
/* We need to make sure that everything we return is on the right
   alignment to hold anything, including a double. */
#define ALIGNMENT sizeof(union alignment)

/* We need to do this cast-to-char* trick on our pointers to adjust
   them; doing arithmetic on a void* is not standard. */
#define OUTPTR(ptr) (((char*)ptr)+ALIGNMENT)
#define INPTR(ptr) (((char*)ptr)-ALIGNMENT)

static size_t total_allocated = 0;
static void *replacement_malloc(size_t sz)
{
    void *chunk = malloc(sz + ALIGNMENT);
    if (!chunk) return chunk;
    total_allocated += sz;
    *(size_t*)chunk = sz;
    return OUTPTR(chunk);
}
static void *replacement_realloc(void *ptr, size_t sz)
{
    size_t old_size = 0;
    if (ptr) {
        ptr = INPTR(ptr);
        old_size = *(size_t*)ptr;
    }
```

```
    ptr = realloc(ptr, sz + ALIGNMENT);
    if (!ptr)
        return NULL;
    *(size_t*)ptr = sz;
    total_allocated = total_allocated - old_size + sz;
    return OUTPTR(ptr);
}
static void replacement_free(void *ptr)
{
    ptr = INPTR(ptr);
    total_allocated -= *(size_t*)ptr;
    free(ptr);
}
void start_counting_bytes(void)
{
    event_set_mem_functions(replacement_malloc,
                            replacement_realloc,
                            replacement_free);
}
```

NOTES

- Replacing the memory management functions affects all future calls to allocate, resize, or free memory from Libevent. Therefore, you need to make sure that you replace the functions *before* you call any other Libevent functions. Otherwise, Libevent will use your version of free to deallocate memory returned from the C library's version of malloc.
- Your malloc and realloc functions need to return memory chunks with the same alignment as the C library.
- Your realloc function needs to handle realloc(NULL, sz) correctly (that is, by treating it as malloc(sz)).
- Your realloc function needs to handle realloc(ptr, 0) correctly (that is, by treating it as free(ptr)).
- Your free function does not need to handle free(NULL).
- Your malloc function does not need to handle malloc(0).
- The replaced memory management functions need to be threadsafe if you are using Libevent from more than one thread.
- Libevent will use these functions to allocate memory that it returns to you. Thus, if you want to free memory that is allocated and returned by a Libevent function, and you have replaced the malloc and realloc functions, then you will probably have to use your replacement free function to free it.

The event_set_mem_functions() function is declared in <event2/event.h>. It first appeared in Libevent 2.0.1-alpha.

Libevent can be built with event_set_mem_functions() disabled. If it is, then programs using event_set_mem_functions will not compile or link. In Libevent 2.0.2-alpha and later, you can detect the presence of event_set_mem_functions() by checking whether the EVENT_SET_MEM_FUNCTIONS_IMPLEMENTED macro is defined.

Locks and threading

As you probably know if you're writing multithreaded programs, it isn't always safe to access the same data from multiple threads at the same time.

Libevent structures can generally work three ways with multithreading.

- Some structures are inherently single-threaded: it is never safe to use them from more than one thread at the same time.
- Some structures are optionally locked: you can tell Libevent for each object whether you need to use it from multiple threads at once.
- Some structures are always locked: if Libevent is running with lock support, then they are always safe to use from multiple threads at once.

To get locking in Libevent, you must tell Libevent which locking functions to use. You need to do this before you call any Libevent function that allocates a structure that needs to be shared between threads.

If you are using the pthreads library, or the native Windows threading code, you're in luck. There are pre-defined functions that will set Libevent up to use the right pthreads or Windows functions for you.

Interface

```
#ifdef WIN32
int evthread_use_windows_threads(void);
#define EVTHREAD_USE_WINDOWS_THREADS_IMPLEMENTED
#endif
#ifdef _EVENT_HAVE_PTHREADS
int evthread_use_pthreads(void);
#define EVTHREAD_USE_PTHREADS_IMPLEMENTED
#endif
```

Both functions return 0 on success, and -1 on failure.

If you need to use a different threading library, then you have a little more work ahead of you. You need to define functions that use your library to implement:

- Locks
- locking
- unlocking
- lock allocation
- lock destruction
- Conditions
- condition variable creation
- condition variable destruction
- waiting on a condition variable
- signaling/broadcasting to a condition variable
- Threads
- thread ID detection

Then you tell Libevent about these functions, using the evthread_set_lock_callbacks and evthread_set_id_callback interfaces.

Interface

```
#define EVTHREAD_WRITE  0x04
#define EVTHREAD_READ   0x08
#define EVTHREAD_TRY    0x10

#define EVTHREAD_LOCKTYPE_RECURSIVE 1
#define EVTHREAD_LOCKTYPE_READWRITE 2

#define EVTHREAD_LOCK_API_VERSION 1

struct evthread_lock_callbacks {
       int lock_api_version;
       unsigned supported_locktypes;
       void *(*alloc)(unsigned locktype);
       void (*free)(void *lock, unsigned locktype);
       int (*lock)(unsigned mode, void *lock);
```

```
        int (*unlock)(unsigned mode, void *lock);
};

int evthread_set_lock_callbacks(const struct evthread_lock_callbacks *);

void evthread_set_id_callback(unsigned long (*id_fn)(void));

struct evthread_condition_callbacks {
        int condition_api_version;
        void *(*alloc_condition)(unsigned condtype);
        void (*free_condition)(void *cond);
        int (*signal_condition)(void *cond, int broadcast);
        int (*wait_condition)(void *cond, void *lock,
            const struct timeval *timeout);
};

int evthread_set_condition_callbacks(
        const struct evthread_condition_callbacks *);
```

The evthread_lock_callbacks structure describes your locking callbacks and their abilities. For the version described above, the lock_api_version field must be set to EVTHREAD_LOCK_API_VERSION. The supported_locktypes field must be set to a bitmask of the EVTHREAD_LOCKTYPE_* constants to describe which lock types you can support. (As of 2.0.4-alpha, EVTHREAD_LOCK_RECURSIVE is mandatory and EVTHREAD_LOCK_READWRITE is unused.) The *alloc* function must return a new lock of the specified type. The *free* function must release all resources held by a lock of the specified type. The *lock* function must try to acquire the lock in the specified mode, returning 0 on success and nonzero on failure. The *unlock* function must try to unlock the lock, returning 0 on success and nonzero on failure.

Recognized lock types are:

0
A regular, not-necessarily recursive lock.

EVTHREAD_LOCKTYPE_RECURSIVE
A lock that does not block a thread already holding it from requiring it again. Other threads can acquire the lock once the thread holding it has unlocked it as many times as it was initially locked.

EVTHREAD_LOCKTYPE_READWRITE
A lock that allows multiple threads to hold it at once for reading, but only one thread at a time to hold it for writing. A writer excludes all readers.

Recognized lock modes are:

EVTHREAD_READ
For READWRITE locks only: acquire or release the lock for reading.

EVTHREAD_WRITE
For READWRITE locks only: acquire or release the lock for writing.

EVTHREAD_TRY
For locking only: acquire the lock only if the lock can be acquired immediately.

The id_fn argument must be a function returning an unsigned long identifying what thread is calling the function. It must always return the same number for the same thread, and must not ever return the same number for two different threads if they are both executing at the same time.

The evthread_condition_callbacks structure describes callbacks related to condition variables. For the version described above, the lock_api_version field must be set to EVTHREAD_CONDITION_API_VERSION. The alloc_condition function must return a pointer to a new condition variable. It receives 0 as its argument. The free_condition function must release storage and resources held by a condition variable. The wait_condition function takes three arguments: a condition allocated by alloc_condition, a lock allocated by the evthread_lock_callbacks.alloc function you provided, and an optional timeout. The lock will be held whenever

the function is called; the function must release the lock, and wait until the condition becomes signalled or until the (optional) timeout has elapsed. The wait_condition function should return -1 on an error, 0 if the condition is signalled, and 1 on a timeout. Before it returns, it should make sure it holds the lock again. Finally, the signal_condition function should cause *one* thread waiting on the condition to wake up (if its broadcast argument is false) and *all* threads currently waiting on the condition to wake up (if its broadcast argument is true). It will only be held while holding the lock associated with the condition.

For more information on condition variables, look at the documentation for pthreads's pthread_cond_* functions, or Windows's CONDITION_VARIABLE functions.

Examples

```
For an example of how to use these functions, see evthread_pthread.c and
evthread_win32.c in the Libevent source distribution.
```

The functions in this section are declared in <event2/thread.h>. Most of them first appeared in Libevent 2.0.4-alpha. Libevent versions from 2.0.1-alpha through 2.0.3-alpha used an older interface to set locking functions. The event_use_pthreads() function requires you to link your program against the event_pthreads library.

The condition-variable functions were new in Libevent 2.0.7-rc; they were added to solve some otherwise intractable deadlock problems.

Libevent can be built with locking support disabled. If it is, then programs built to use the above thread-related functions will not run.

Debugging lock usage

To help debug lock usage, Libevent has an optional "lock debugging" feature that wraps its locking calls in order to catch typical lock errors, including:

- unlocking a lock that we don't actually hold
- re-locking a non-recursive lock

If one of these lock errors occurs, Libevent exits with an assertion failure.

Interface

```
void evthread_enable_lock_debugging(void);
#define evthread_enable_lock_debuging() evthread_enable_lock_debugging()
```

Note
This function MUST be called before any locks are created or used. To be safe, call it just after you set your threading functions.

This function was new in Libevent 2.0.4-alpha with the misspelled name "evthread_enable_lock_debuging()." The spelling was fixed to evthread_enable_lock_debugging() in 2.1.2-alpha; both names are currently supported.

Debugging event usage

There are some common errors in using events that Libevent can detect and report for you. They include:

- Treating an uninitialized struct event as though it were initialized.
- Try to reinitialize a pending struct event.

Tracking which events are initialized requires that Libevent use extra memory and CPU, so you should only enable debug mode when actually debugging your program.

Interface

```
void event_enable_debug_mode(void);
```

This function must only be called before any event_base is created.

When using debug mode, you might run out of memory if your program uses a large number of events created with event_assign() [not event_new()]. This happens because Libevent has no way of telling when an event created with event_assign() will no longer be used. (It can tell that an event_new() event has become invalid when you call event_free() on it.) If you want to avoid running out of memory while debugging, you can explicitly tell Libevent that such events are no longer to be treated as assigned:

Interface

```
void event_debug_unassign(struct event *ev);
```

Calling event_debug_unassign() has no effect when debugging is not enabled.

Example

```
#include <event2/event.h>
#include <event2/event_struct.h>

#include <stdlib.h>

void cb(evutil_socket_t fd, short what, void *ptr)
{
    /* We pass 'NULL' as the callback pointer for the heap allocated
     * event, and we pass the event itself as the callback pointer
     * for the stack-allocated event. */
    struct event *ev = ptr;

    if (ev)
        event_debug_unassign(ev);
}

/* Here's a simple mainloop that waits until fd1 and fd2 are both
 * ready to read. */
void mainloop(evutil_socket_t fd1, evutil_socket_t fd2, int debug_mode)
{
    struct event_base *base;
    struct event event_on_stack, *event_on_heap;

    if (debug_mode)
       event_enable_debug_mode();

    base = event_base_new();

    event_on_heap = event_new(base, fd1, EV_READ, cb, NULL);
    event_assign(&event_on_stack, base, fd2, EV_READ, cb, &event_on_stack);

    event_add(event_on_heap, NULL);
    event_add(&event_on_stack, NULL);

    event_base_dispatch(base);

    event_free(event_on_heap);
    event_base_free(base);
}
```

Detailed event debugging is a feature which can only be enabled at compile-time using the CFLAGS environment variable "-DUSE_DEBUG". With this flag enabled, any program compiled against Libevent will output a very verbose log detailing low-level activity on the back-end. These logs include, but not limited to, the following:

- event additions
- event deletions
- platform specific event notification information

This feature cannot be enabled or disabled via an API call so it must only be used in developer builds.

These debugging functions were added in Libevent 2.0.4-alpha.

Detecting the version of Libevent

New versions of Libevent can add features and remove bugs. Sometimes you'll want to detect the Libevent version, so that you can:

- Detect whether the installed version of Libevent is good enough to build your program.
- Display the Libevent version for debugging.
- Detect the version of Libevent so that you can warn the user about bugs, or work around them.

Interface

```
#define LIBEVENT_VERSION_NUMBER 0x02000300
#define LIBEVENT_VERSION "2.0.3-alpha"
const char *event_get_version(void);
ev_uint32_t event_get_version_number(void);
```

The macros make available the compile-time version of the Libevent library; the functions return the run-time version. Note that if you have dynamically linked your program against Libevent, these versions may be different.

You can get a Libevent version in two formats: as a string suitable for displaying to users, or as a 4-byte integer suitable for numerical comparison. The integer format uses the high byte for the major version, the second byte for the minor version, the third byte for the patch version, and the low byte to indicate release status (0 for release, nonzero for a development series after a given release).

Thus, the released Libevent 2.0.1-alpha has the version number of [02 00 01 00], or 0x02000100. A development versions between 2.0.1-alpha and 2.0.2-alpha might have a version number of [02 00 01 08], or 0x02000108.

Example: Compile-time checks

```
#include <event2/event.h>

#if !defined(LIBEVENT_VERSION_NUMBER) || LIBEVENT_VERSION_NUMBER < 0x02000100
#error "This version of Libevent is not supported; Get 2.0.1-alpha or later."
#endif

int
make_sandwich(void)
{
        /* Let's suppose that Libevent 6.0.5 introduces a make-me-a
           sandwich function. */
#if LIBEVENT_VERSION_NUMBER >= 0x06000500
        evutil_make_me_a_sandwich();
        return 0;
#else
        return -1;
#endif
}
```

Example: Run-time checks

```
#include <event2/event.h>
#include <string.h>

int
check_for_old_version(void)
{
    const char *v = event_get_version();
    /* This is a dumb way to do it, but it is the only thing that works
       before Libevent 2.0. */
    if (!strncmp(v, "0.", 2) ||
        !strncmp(v, "1.1", 3) ||
        !strncmp(v, "1.2", 3) ||
        !strncmp(v, "1.3", 3)) {

        printf("Your version of Libevent is very old.  If you run into bugs,"
               " consider upgrading.\n");
        return -1;
    } else {
        printf("Running with Libevent version %s\n", v);
        return 0;
    }
}

int
check_version_match(void)
{
    ev_uint32_t v_compile, v_run;
    v_compile = LIBEVENT_VERSION_NUMBER;
    v_run = event_get_version_number();
    if ((v_compile & 0xffff0000) != (v_run & 0xffff0000)) {
        printf("Running with a Libevent version (%s) very different from the "
               "one we were built with (%s).\n", event_get_version(),
               LIBEVENT_VERSION);
        return -1;
    }
    return 0;
}
```

The macros and functions in this section are defined in <event2/event.h>. The event_get_version() function first appeared in Libevent 1.0c; the others first appeared in Libevent 2.0.1-alpha.

Freeing global Libevent structures

Even when you've freed all the objects that you allocated with Libevent, there will be a few globally allocated structures left over. This isn't usually a problem: once the process exits, they will all get cleaned up anyway. But having these structures can confuse some debugging tools into thinking that Libevent is leaking resources. If you need to make sure that Libevent has released all internal library-global data structures, you can call:

Interface

```
void libevent_global_shutdown(void);
```

This function doesn't free any structures that were returned to you by a Libevent function. If you want to free everything before exiting, you'll need to free all events, event_bases, bufferevents, and so on yourself.

Calling libevent_global_shutdown() will make other Libevent functions behave unpredictably; don't call it except as the last Libevent function your program invokes. One exception is that libevent_global_shutdown() is idempotent: it is okay to call it even if it has already been called.

This function is declared in <event2/event.h>. It was introduced in Libevent 2.1.1-alpha.

Creating an event_base

Before you can use any interesting Libevent function, you need to allocate one or more event_base structures. Each event_base structure holds a set of events and can poll to determine which events are active.

If an event_base is set up to use locking, it is safe to access it between multiple threads. Its loop can only be run in a single thread, however. If you want to have multiple threads polling for IO, you need to have an event_base for each thread.

Tip
[A future version of Libevent may have support for event_bases that run events across multiple threads.]

Each event_base has a "method", or a backend that it uses to determine which events are ready. The recognized methods are:

- select
- poll
- epoll
- kqueue
- devpoll
- evport
- win32

The user can disable specific backends with environment variables. If you want to turn off the kqueue backend, set the EVENT_NOKQUEUE environment variable, and so on. If you want to turn off backends from within the program, see notes on event_config_avoid_method() below.

Setting up a default event_base

The event_base_new() function allocates and returns a new event base with the default settings. It examines the environment variables and returns a pointer to a new event_base. If there is an error, it returns NULL.

When choosing among methods, it picks the fastest method that the OS supports.

Interface

```
struct event_base *event_base_new(void);
```

For most programs, this is all you need.

The event_base_new() function is declared in <event2/event.h>. It first appeared in Libevent 1.4.3.

Setting up a complicated event_base

If you want more control over what kind of event_base you get, you need to use an event_config. An event_config is an opaque structure that holds information about your preferences for an event_base. When you want an event_base, you pass the event_config to event_base_new_with_config().

Interface

```
struct event_config *event_config_new(void);
struct event_base *event_base_new_with_config(const struct event_config *cfg);
void event_config_free(struct event_config *cfg);
```

To allocate an event_base with these functions, you call event_config_new() to allocate a new event_config. Then, you call other functions on the event_config to tell it about your needs. Finally, you call event_base_new_with_config() to get a new event_base. When you are done, you can free the event_config with event_config_free().

Interface

```
int event_config_avoid_method(struct event_config *cfg, const char *method);

enum event_method_feature {
    EV_FEATURE_ET = 0x01,
    EV_FEATURE_O1 = 0x02,
    EV_FEATURE_FDS = 0x04,
};
int event_config_require_features(struct event_config *cfg,
                                  enum event_method_feature feature);

enum event_base_config_flag {
    EVENT_BASE_FLAG_NOLOCK = 0x01,
    EVENT_BASE_FLAG_IGNORE_ENV = 0x02,
    EVENT_BASE_FLAG_STARTUP_IOCP = 0x04,
    EVENT_BASE_FLAG_NO_CACHE_TIME = 0x08,
    EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST = 0x10,
    EVENT_BASE_FLAG_PRECISE_TIMER = 0x20
};
int event_config_set_flag(struct event_config *cfg,
    enum event_base_config_flag flag);
```

Calling event_config_avoid_method tells Libevent to avoid a specific available backend by name. Calling event_config_require_feature() tells Libevent not to use any backend that cannot supply all of a set of features. Calling event_config_set_flag() tells Libevent to set one or more of the run-time flags below when constructing the event base.

The recognized feature values for event_config_require_features are:

EV_FEATURE_ET
: Requires a backend method that supports edge-triggered IO.

EV_FEATURE_O1
: Requires a backend method where adding or deleting a single event, or having a single event become active, is an O(1) operation.

EV_FEATURE_FDS
: Requires a backend method that can support arbitrary file descriptor types, and not just sockets.

The recognized option values for event_config_set_flag() are:

EVENT_BASE_FLAG_NOLOCK
: Do not allocate locks for the event_base. Setting this option may save a little time for locking and releasing the event_base, but will make it unsafe and nonfunctional to access it from multiple threads.

EVENT_BASE_FLAG_IGNORE_ENV
Do not check the EVENT_* environment variables when picking which backend method to use. Think hard before using this flag: it can make it harder for users to debug the interactions between your program and Libevent.

EVENT_BASE_FLAG_STARTUP_IOCP
On Windows only, this flag makes Libevent enable any necessary IOCP dispatch logic on startup, rather than on-demand.

EVENT_BASE_FLAG_NO_CACHE_TIME
Instead of checking the current time every time the event loop is ready to run timeout callbacks, check it after every timeout callback. This can use more CPU than you necessarily intended, so watch out!

EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST
Tells Libevent that, if it decides to use the epoll backend, it is safe to use the faster "changelist"-based backend. The epoll-changelist backend can avoid needless system calls in cases where the same fd has its status modified more than once between calls to the backend's dispatch function, but it also trigger a kernel bug that causes erroneous results if you give Libevent any fds cloned by dup() or its variants. This flag has no effect if you use a backend other than epoll. You can also turn on the epoll-changelist option by setting the EVENT_EPOLL_USE_CHANGELIST environment variable.

EVENT_BASE_FLAG_PRECISE_TIMER
By default, Libevent tries to use the fastest available timing mechanism that the operating system provides. If there is a slower timing mechanism that provides more fine-grained timing precision, this flag tells Libevent to use that timing mechanism instead. If the operating system provides no such slower-but-more-precise mechanism, this flag has no effect.

The above functions that manipulate an event_config all return 0 on success, -1 on failure.

Note
It is easy to set up an event_config that requires a backend that your OS does not provide. For example, as of Libevent 2.0.1-alpha, there is no O(1) backend for Windows, and no backend on Linux that provides both EV_FEATURE_FDS and EV_FEATURE_O1. If you have made a configuration that Libevent can't satisfy, event_base_new_with_config() will return NULL.

Interface

```
int event_config_set_num_cpus_hint(struct event_config *cfg, int cpus)
```

This function is currently only useful with Windows when using IOCP, though it may become useful for other platforms in the future. Calling it tells the event_config that the event_base it generates should try to make good use of a given number of CPUs when multithreading. Note that this is only a hint: the event base may wind up using more or fewer CPUs than you select.

Interface

```
int event_config_set_max_dispatch_interval(struct event_config *cfg,
    const struct timeval *max_interval, int max_callbacks,
    int min_priority);
```

This function prevents priority inversion by limiting how many low-priority event callbacks can be invoked before checking for more high-priority events. If max_interval is non-null, the event loop checks the time after each callback, and re-scans for high-priority events if max_interval has passed. If max_callbacks is nonnegative, the event loop also checks for more events after max_callbacks callbacks have been invoked. These rules apply to any event of min_priority or higher.

Example: Preferring edge-triggered backends

```
struct event_config *cfg;
struct event_base *base;
int i;

/* My program wants to use edge-triggered events if at all possible.  So
   I'll try to get a base twice: Once insisting on edge-triggered IO, and
   once not. */
for (i=0; i<2; ++i) {
```

```
    cfg = event_config_new();

    /* I don't like select. */
    event_config_avoid_method(cfg, "select");

    if (i == 0)
        event_config_require_features(cfg, EV_FEATURE_ET);

    base = event_base_new_with_config(cfg);
    event_config_free(cfg);
    if (base)
        break;

    /* If we get here, event_base_new_with_config() returned NULL.  If
       this is the first time around the loop, we'll try again without
       setting EV_FEATURE_ET.  If this is the second time around the
       loop, we'll give up. */
}
```

Example: Avoiding priority-inversion

```
struct event_config *cfg;
struct event_base *base;

cfg = event_config_new();
if (!cfg)
   /* Handle error */;

/* I'm going to have events running at two priorities.  I expect that
   some of my priority-1 events are going to have pretty slow callbacks,
   so I don't want more than 100 msec to elapse (or 5 callbacks) before
   checking for priority-0 events. */
struct timeval msec_100 = { 0, 100*1000 };
event_config_set_max_dispatch_interval(cfg, &msec_100, 5, 1);

base = event_base_new_with_config(cfg);
if (!base)
   /* Handle error */;

event_base_priority_init(base, 2);
```

These functions and types are declared in <event2/event.h>.

The EVENT_BASE_FLAG_IGNORE_ENV flag first appeared in Libevent 2.0.2-alpha. The EVENT_BASE_FLAG_PRECISE_TIMER flag first appeared in Libevent 2.1.2-alpha. The event_config_set_num_cpus_hint() function was new in Libevent 2.0.7-rc, and event_config_set_max_dispatch_interval() was new in 2.1.1-alpha. Everything else in this section first appeared in Libevent 2.0.1-alpha.

Examining an event_base's backend method

Sometimes you want to see which features are actually available in an event_base, or which method it's using.

Interface

```
const char **event_get_supported_methods(void);
```

The event_get_supported_methods() function returns a pointer to an array of the names of the methods supported in this version of Libevent. The last element in the array is NULL.

Example

```
int i;
const char **methods = event_get_supported_methods();
printf("Starting Libevent %s.  Available methods are:\n",
    event_get_version());
for (i=0; methods[i] != NULL; ++i) {
    printf("    %s\n", methods[i]);
}
```

Note
This function returns a list of the methods that Libevent was compiled to support. It is possible that your operating system will not in fact support them all when Libevent tries to run. For example, you could be on a version of OSX where kqueue is too buggy to use.

Interface

```
const char *event_base_get_method(const struct event_base *base);
enum event_method_feature event_base_get_features(const struct event_base *base);
```

The event_base_get_method() call returns the name of the actual method in use by an event_base. The event_base_get_features() call returns a bitmask of the features that it supports.

Example

```
struct event_base *base;
enum event_method_feature f;

base = event_base_new();
if (!base) {
    puts("Couldn't get an event_base!");
} else {
    printf("Using Libevent with backend method %s.",
        event_base_get_method(base));
    f = event_base_get_features(base);
    if ((f & EV_FEATURE_ET))
        printf("  Edge-triggered events are supported.");
    if ((f & EV_FEATURE_O1))
        printf("  O(1) event notification is supported.");
    if ((f & EV_FEATURE_FDS))
        printf("  All FD types are supported.");
    puts("");
}
```

These functions are defined in <event2/event.h>. The event_base_get_method() call was first available in Libevent 1.4.3. The others first appeared in Libevent 2.0.1-alpha.

Deallocating an event_base

When you are finished with an event_base, you can deallocate it with event_base_free().

Interface

```
void event_base_free(struct event_base *base);
```

Note that this function does not deallocate any of the events that are currently associated with the event_base, or close any of their sockets, or free any of their pointers.

The event_base_free() function is defined in <event2/event.h>. It was first implemented in Libevent 1.2.

Setting priorities on an event_base

Libevent supports setting multiple priorities on an event. By default, though, an event_base supports only a single priority level. You can set the number of priorities on an event_base by calling event_base_priority_init().

Interface

```
int event_base_priority_init(struct event_base *base, int n_priorities);
```

This function returns 0 on success and -1 on failure. The *base* argument is the event_base to modify, and n_priorities is the number of priorities to support. It must be at least 1. The available priorities for new events will be numbered from 0 (most important) to n_priorities-1 (least important).

There is a constant, EVENT_MAX_PRIORITIES, that sets the upper bound on the value of n_priorities. It is an error to call this function with a higher value for n_priorities.

Note
You **must** call this function before any events become active. It is best to call it immediately after creating the event_base.

To find the number of priorities currently supported by a base, you can call event_base_getnpriorities().

Interface

```
int event_base_get_npriorities(struct event_base *base);
```

The return value is equal to the number of priorities configured in the base. So if event_base_get_npriorities() returns 3, then allowable priority values are 0, 1, and 2.

Example

```
For an example, see the documentation for event_priority_set below.
```

By default, all new events associated with this base will be initialized with priority equal to n_priorities / 2.

The event_base_priority_init function is defined in <event2/event.h>. It has been available since Libevent 1.0. The event_base_get_nprio function was new in Libevent 2.1.1-alpha.

Reinitializing an event_base after fork()

Not all event backends persist cleanly after a call to fork(). Thus, if your program uses fork() or a related system call in order to start a new process, and you want to continue using an event_base after you have forked, you may need to reinitialize it.

Interface

```
int event_reinit(struct event_base *base);
```

The function returns 0 on success, -1 on failure.

Example

```
struct event_base *base = event_base_new();

/* ... add some events to the event_base ... */

if (fork()) {
    /* In parent */
    continue_running_parent(base); /*...*/
} else {
    /* In child */
    event_reinit(base);
    continue_running_child(base); /*...*/
}
```

The event_reinit() function is defined in <event2/event.h>. It was first available in Libevent 1.4.3-alpha.

Obsolete event_base functions

Older versions of Libevent relied pretty heavily on the idea of a "current" event_base. The "current" event_base was a global setting shared across all threads. If you forgot to specify which event_base you wanted, you got the current one. Since event_bases weren't threadsafe, this could get pretty error-prone.

Instead of event_base_new(), there was:

Interface

```
struct event_base *event_init(void);
```

This function worked like event_base_new(), and set the current base to the allocated base. There was no other way to change the current base.

Some of the event_base functions in this section had variants that operated on the current base. These functions behaved as the current functions, except that they took no base argument.

Current function	Obsolete current-base version
event_base_priority_init()	event_priority_init()
event_base_get_method()	event_get_method()

Working with an event loop

Running the loop

Once you have an event_base with some events registered (see the next section about how to create and register events), you will want Libevent to wait for events and alert you about them.

Interface

```
#define EVLOOP_ONCE             0x01
#define EVLOOP_NONBLOCK         0x02
#define EVLOOP_NO_EXIT_ON_EMPTY 0x04

int event_base_loop(struct event_base *base, int flags);
```

By default, the event_base_loop() function *runs* an event_base until there are no more events registered in it. To run the loop, it repeatedly checks whether any of the registered events has triggered (for example, if a read event's file descriptor is ready to read, or if a timeout event's timeout is ready to expire). Once this happens, it marks all triggered events as "active", and starts to run them.

You can change the behavior of event_base_loop() by setting one or more flags in its *flags* argument. If EVLOOP_ONCE is set, then the loop will wait until some events become active, then run active events until there are no more to run, then return. If EVLOOP_NONBLOCK is set, then the loop will not wait for events to trigger: it will only check whether any events are ready to trigger immediately, and run their callbacks if so.

Ordinarily, the loop will exit as soon as it has no pending or active events. You can override this behavior by passing the EVLOOP_NO_EXIT_ON_EMPTY flag---for example, if you're going to be adding events from some other thread. If you

do set EVLOOP_NO_EXIT_ON_EMPTY, the loop will keep running until somebody calls event_base_loopbreak(), or calls event_base_loopexit(), or an error occurs.

When it is done, event_base_loop() returns 0 if it exited normally, -1 if it exited because of some unhandled error in the backend, and 1 if it exited because there were no more pending or active events.

To aid in understanding, here's an approximate summary of the event_base_loop algorithm:

Pseudocode

```
while (any events are registered with the loop,
        or EVLOOP_NO_EXIT_ON_EMPTY was set) {

    if (EVLOOP_NONBLOCK was set, or any events are already active)
        If any registered events have triggered, mark them active.
    else
        Wait until at least one event has triggered, and mark it active.

    for (p = 0; p < n_priorities; ++p) {
       if (any event with priority of p is active) {
          Run all active events with priority of p.
          break; /* Do not run any events of a less important priority */
       }
    }

    if (EVLOOP_ONCE was set or EVLOOP_NONBLOCK was set)
       break;
}
```

As a convenience, you can also call:

Interface

```
int event_base_dispatch(struct event_base *base);
```

The event_base_dispatch() call is the same as event_base_loop(), with no flags set. Thus, it keeps running until there are no more registered events or until event_base_loopbreak() or event_base_loopexit() is called.

These functions are defined in <event2/event.h>. They have existed since Libevent 1.0.

Stopping the loop

If you want an active event loop to stop running before all events are removed from it, you have two slightly different functions you can call.

Interface

```
int event_base_loopexit(struct event_base *base,
                        const struct timeval *tv);
int event_base_loopbreak(struct event_base *base);
```

The event_base_loopexit() function tells an event_base to stop looping after a given time has elapsed. If the *tv* argument is NULL, the event_base stops looping without a delay. If the event_base is currently running callbacks for any active events, it will continue running them, and not exit until they have all been run.

The event_base_loopbreak() function tells the event_base to exit its loop immediately. It differs from event_base_loopexit(base, NULL) in that if the event_base is currently running callbacks for any active events, it will exit immediately after finishing the one it's currently processing.

Note also that event_base_loopexit(base,NULL) and event_base_loopbreak(base) act differently when no event loop is running: loopexit schedules the next instance of the event loop to stop right after the next round of callbacks are run (as if it had been invoked with EVLOOP_ONCE) whereas loopbreak only stops a currently running loop, and has no effect if the event loop isn't running.

Both of these methods return 0 on success and -1 on failure.

Example: Shut down immediately

```
#include <event2/event.h>

/* Here's a callback function that calls loopbreak */
void cb(int sock, short what, void *arg)
{
    struct event_base *base = arg;
    event_base_loopbreak(base);
}

void main_loop(struct event_base *base, evutil_socket_t watchdog_fd)
{
    struct event *watchdog_event;

    /* Construct a new event to trigger whenever there are any bytes to
       read from a watchdog socket.  When that happens, we'll call the
       cb function, which will make the loop exit immediately without
       running any other active events at all.
     */
    watchdog_event = event_new(base, watchdog_fd, EV_READ, cb, base);

    event_add(watchdog_event, NULL);

    event_base_dispatch(base);
}
```

Example: Run an event loop for 10 seconds, then exit.

```
#include <event2/event.h>

void run_base_with_ticks(struct event_base *base)
{
  struct timeval ten_sec;

  ten_sec.tv_sec = 10;
  ten_sec.tv_usec = 0;

  /* Now we run the event_base for a series of 10-second intervals, printing
     "Tick" after each.  For a much better way to implement a 10-second
     timer, see the section below about persistent timer events. */
  while (1) {
     /* This schedules an exit ten seconds from now. */
     event_base_loopexit(base, &ten_sec);

     event_base_dispatch(base);
     puts("Tick");
  }
}
```

Sometimes you may want to tell whether your call to event_base_dispatch() or event_base_loop() exited normally, or because of a call to event_base_loopexit() or event_base_break(). You can use these functions to tell whether loopexit or break was called:

Interface

```
int event_base_got_exit(struct event_base *base);
int event_base_got_break(struct event_base *base);
```

These two functions will return true if the loop was stopped with event_base_loopexit() or event_base_break() respectively, and false otherwise. Their values will be reset the next time you start the event loop.

These functions are declared in <event2/event.h>. The event_break_loopexit() function was first implemented in Libevent 1.0c; event_break_loopbreak() was first implemented in Libevent 1.4.3.

Re-checking for events

Ordinarily, Libevent checks for events, then runs all the active events with the highest priority, then checks for events again, and so on. But sometimes you might want to stop Libevent right after the current callback has been run, and tell it to scan again. By analogy to event_base_loopbreak(), you can do this with the function event_base_loopcontinue().

Interface

```
int event_base_loopcontinue(struct event_base *);
```

Calling event_base_loopcontinue() has no effect if we aren't currently running event callbacks.

This function was introduced in Libevent 2.1.2-alpha.

Checking the internal time cache

Sometimes you want to get an approximate view of the current time inside an event callback, and you want to get it without calling gettimeofday() yourself (presumably because your OS implements gettimeofday() as a syscall, and you're trying to avoid syscall overhead).

From within a callback, you can ask Libevent for its view of the current time when it began executing this round of callbacks:

Interface

```
int event_base_gettimeofday_cached(struct event_base *base,
    struct timeval *tv_out);
```

The event_base_gettimeofday_cached() function sets the value of its *tv_out* argument to the cached time if the event_base is currently executing callbacks. Otherwise, it calls evutil_gettimeofday() for the actual current time. It returns 0 on success, and negative on failure.

Note that since the timeval is cached when Libevent starts running callbacks, it will be at least a little inaccurate. If your callbacks take a long time to run, it may be **very** inaccurate. To force an immediate cache update, you can call this function:

Interface

```
int event_base_update_cache_time(struct event_base *base);
```

It returns 0 on success and -1 on failure, and has no effect if the base was not running its event loop.

The event_base_gettimeofday_cached() function was new in Libevent 2.0.4-alpha. Libevent 2.1.1-alpha added event_base_update_cache

Dumping the event_base status

Interface

```
void event_base_dump_events(struct event_base *base, FILE *f);
```

For help debugging your program (or debugging Libevent!) you might sometimes want a complete list of all events added in the event_base and their status. Calling event_base_dump_events() writes this list to the stdio file provided.

The list is meant to be human-readable; its format **will** change in future versions of Libevent.

This function was introduced in Libevent 2.0.1-alpha.

Running a function over every event in an event_base

Interface

```
typedef int (*event_base_foreach_event_cb)(const struct event_base *,
    const struct event *, void *);

int event_base_foreach_event(struct event_base *base,
                             event_base_foreach_event_cb fn,
                             void *arg);
```

You can use event_base_foreach_event() to iterate over every currently active or pending event associated with an event_base(). The provided callback will be invoked exactly once per event, in an unspecified order. The third argument of event_base_foreach_event() will be passed as the third argument to each invocation of the callback.

The callback function must return 0 to continue iteration, or some other integer to stop iterating. Whatever value the callback function finally returns will then be returned by event_base_foreach_function().

Your callback function **must not** modify any of the events that it receives, or add or remove any events to the event base, or otherwise modify any event associated with the event base, or undefined behavior can occur, up to or including crashes and heap-smashing.

The event_base lock will be held for the duration of the call to event_base_foreach_event() — this will block other threads from doing anything useful with the event_base, so make sure that your callback doesn't take a long time.

This function was added in Libevent 2.1.2-alpha.

Obsolete event loop functions

As discussed above, older versions of Libevent APIs had a global notion of a "current" event_base.

Some of the event loop functions in this section had variants that operated on the current base. These functions behaved as the current functions, except that they took no base argument.

Current function	Obsolete current-base version
event_base_dispatch()	event_dispatch()
event_base_loop()	event_loop()
event_base_loopexit()	event_loopexit()
event_base_loopbreak()	event_loopbreak()

Note
Because event_base did not support locking before Libevent 2.0, these functions weren't completely threadsafe: it was not permissible to call the _loopbreak() or _loopexit() functions from a thread other than the one executing the event loop.

Working with events

Libevent's basic unit of operation is the *event*. Every event represents a set of conditions, including:

- A file descriptor being ready to read from or write to.
- A file descriptor *becoming* ready to read from or write to (Edge-triggered IO only).
- A timeout expiring.
- A signal occurring.
- A user-triggered event.

Events have similar lifecycles. Once you call a Libevent function to set up an event and associate it with an event base, it becomes **initialized**. At this point, you can *add*, which makes it **pending** in the base. When the event is pending, if the conditions that would trigger an event occur (e.g., its file descriptor changes state or its timeout expires), the event becomes **active**, and its (user-provided) callback function is run. If the event is configured **persistent**, it remains pending. If it is not persistent, it stops being pending when its callback runs. You can make a pending event non-pending by *deleting* it, and you can *add* a non-pending event to make it pending again.

Constructing event objects

To create a new event, use the event_new() interface.

Interface

```
#define EV_TIMEOUT      0x01
#define EV_READ         0x02
#define EV_WRITE        0x04
#define EV_SIGNAL       0x08
#define EV_PERSIST      0x10
#define EV_ET           0x20

typedef void (*event_callback_fn)(evutil_socket_t, short, void *);

struct event *event_new(struct event_base *base, evutil_socket_t fd,
    short what, event_callback_fn cb,
    void *arg);

void event_free(struct event *event);
```

The event_new() function tries to allocate and construct a new event for use with *base*. The *what* argument is a set of the flags listed above. (Their semantics are described below.) If *fd* is nonnegative, it is the file that we'll observe for read or write events. When the event is active, Libevent will invoke the provided *cb* function, passing it as arguments: the file descriptor *fd*, a bitfield of *all* the events that triggered, and the value that was passed in for *arg* when the function was constructed.

On an internal error, or invalid arguments, event_new() will return NULL.

All new events are initialized and non-pending. To make an event pending, call event_add() (documented below).

To deallocate an event, call event_free(). It is safe to call event_free() on an event that is pending or active: doing so makes the event non-pending and inactive before deallocating it.

Example

```
#include <event2/event.h>

void cb_func(evutil_socket_t fd, short what, void *arg)
{
        const char *data = arg;
        printf("Got an event on socket %d:%s%s%s%s [%s]",
            (int) fd,
            (what&EV_TIMEOUT) ? " timeout" : "",
            (what&EV_READ)    ? " read" : "",
            (what&EV_WRITE)   ? " write" : "",
            (what&EV_SIGNAL)  ? " signal" : "",
```

```
            data);
}

void main_loop(evutil_socket_t fd1, evutil_socket_t fd2)
{
        struct event *ev1, *ev2;
        struct timeval five_seconds = {5,0};
        struct event_base *base = event_base_new();

        /* The caller has already set up fd1, fd2 somehow, and make them
           nonblocking. */

        ev1 = event_new(base, fd1, EV_TIMEOUT|EV_READ|EV_PERSIST, cb_func,
           (char*)"Reading event");
        ev2 = event_new(base, fd2, EV_WRITE|EV_PERSIST, cb_func,
           (char*)"Writing event");

        event_add(ev1, &five_seconds);
        event_add(ev2, NULL);
        event_base_dispatch(base);
}
```

The above functions are defined in <event2/event.h>, and first appeared in Libevent 2.0.1-alpha. The event_callback_fn type first appeared as a typedef in Libevent 2.0.4-alpha.

The event flags

EV_TIMEOUT
This flag indicates an event that becomes active after a timeout elapses.

```
The EV_TIMEOUT flag is ignored when constructing an event: you
can either set a timeout when you add the event, or not.  It is
set in the 'what' argument to the callback function when a timeout
has occurred.
```

EV_READ
This flag indicates an event that becomes active when the provided file descriptor is ready for reading.

EV_WRITE
This flag indicates an event that becomes active when the provided file descriptor is ready for writing.

EV_SIGNAL
Used to implement signal detection. See "Constructing signal events" below.

EV_PERSIST
Indicates that the event is *persistent*. See "About Event Persistence" below.

EV_ET
Indicates that the event should be edge-triggered, if the underlying event_base backend supports edge-triggered events. This affects the semantics of EV_READ and EV_WRITE.

Since Libevent 2.0.1-alpha, any number of events may be pending for the same conditions at the same time. For example, you may have two events that will become active if a given fd becomes ready to read. The order in which their callbacks are run is undefined.

These flags are defined in <event2/event.h>. All have existed since before Libevent 1.0, except for EV_ET, which was introduced in Libevent 2.0.1-alpha.

About Event Persistence

By default, whenever a pending event becomes active (because its fd is ready to read or write, or because its timeout expires), it becomes non-pending right before its callback is executed. Thus, if you want to make the event pending again, you can call event_add() on it again from inside the callback function.

If the EV_PERSIST flag is set on an event, however, the event is *persistent.* This means that event remains pending even when its callback is activated. If you want to make it non-pending from within its callback, you can call event_del() on it.

The timeout on a persistent event resets whenever the event's callback runs. Thus, if you have an event with flags EV_READ|EV_PERSIS and a timeout of five seconds, the event will become active:

- Whenever the socket is ready for reading.
- Whenever five seconds have passed since the event last became active.

Creating an event as its own callback argument

Frequently, you might want to create an event that receives itself as a callback argument. You can't just pass a pointer to the event as an argument to event_new(), though, because it does not exist yet. To solve this problem, you can use event_self_cbarg().

Interface

```
void *event_self_cbarg();
```

The event_self_cbarg() function returns a "magic" pointer which, when passed as an event callback argument, tells event_new() to create an event receiving itself as its callback argument.

Example

```
#include <event2/event.h>

static int n_calls = 0;

void cb_func(evutil_socket_t fd, short what, void *arg)
{
    struct event *me = arg;

    printf("cb_func called %d times so far.\n", ++n_calls);

    if (n_calls > 100)
       event_del(me);
}

void run(struct event_base *base)
{
    struct timeval one_sec = { 1, 0 };
    struct event *ev;
    /* We're going to set up a repeating timer to get called called 100
       times. */
    ev = event_new(base, -1, EV_PERSIST, cb_func, event_self_cbarg());
    event_add(ev, &one_sec);
    event_base_dispatch(base);
}
```

This function can also be used with event_new(), evtimer_new(), evsignal_new(), event_assign(), evtimer_assign(), and evsignal_assign(). It won't work as a callback argument for non-events, however.

The event_self_cbarg() function was introduced in Libevent 2.1.1-alpha.

Timeout-only events

As a convenience, there are a set of macros beginning with evtimer_ that you can use in place of the event_* calls to allocate and manipulate pure-timeout events. Using these macros provides no benefit beyond improving the clarity of your code.

Interface

```
#define evtimer_new(base, callback, arg) \
    event_new((base), -1, 0, (callback), (arg))
#define evtimer_add(ev, tv) \
    event_add((ev),(tv))
#define evtimer_del(ev) \
    event_del(ev)
#define evtimer_pending(ev, tv_out) \
    event_pending((ev), EV_TIMEOUT, (tv_out))
```

These macros have been present since Libevent 0.6, except for evtimer_new(), which first appeared in Libevent 2.0.1-alpha.

Constructing signal events

Libevent can also watch for POSIX-style signals. To construct a handler for a signal, use:

Interface

```
#define evsignal_new(base, signum, cb, arg) \
    event_new(base, signum, EV_SIGNAL|EV_PERSIST, cb, arg)
```

The arguments are as for event_new, except that we provide a signal number instead of a file descriptor.

Example

```
struct event *hup_event;
struct event_base *base = event_base_new();

/* call sighup_function on a HUP signal */
hup_event = evsignal_new(base, SIGHUP, sighup_function, NULL);
```

Note that signal callbacks are run in the event loop after the signal occurs, so it is safe for them to call functions that you are not supposed to call from a regular POSIX signal handler.

Warning
Don't set a timeout on a signal event. It might not be supported. [FIXME: is this true?]

There are also a set of convenience macros you can use when working with signal events.

Interface

```
#define evsignal_add(ev, tv) \
    event_add((ev),(tv))
#define evsignal_del(ev) \
    event_del(ev)
#define evsignal_pending(ev, what, tv_out) \
    event_pending((ev), (what), (tv_out))
```

The evsignal_* macros have been present since Libevent 2.0.1-alpha. Prior versions called them signal_add(), signal_del(), and so on.

Caveats when working with signals

With current versions of Libevent, with most backends, only one event_base per process at a time can be listening for signals. If you add signal events to two event_bases at once ---even if the signals are different!--- only one event_base will receive signals.

The kqueue backend does not have this limitation.

Setting up events without heap-allocation

For performance and other reasons, some people like to allocate events as a part of a larger structure. For each use of the event, this saves them:

- The memory allocator overhead for allocating a small object on the heap.
- The time overhead for dereferencing the pointer to the struct event.
- The time overhead from a possible additional cache miss if the event is not already in the cache.

Using this method risks breaking binary compatibility with other versions of of Libevent, which may have different sizes for the event structure.

These are *very* small costs, and do not matter for most applications. You should just stick to using event_new() unless you **know** that you're incurring a significant performance penalty for heap-allocating your events. Using event_assign() can cause hard-to-diagnose errors with future versions of Libevent if they use a larger event structure than the one you're building with.

Interface

```
int event_assign(struct event *event, struct event_base *base,
    evutil_socket_t fd, short what,
    void (*callback)(evutil_socket_t, short, void *), void *arg);
```

All the arguments of event_assign() are as for event_new(), except for the *event* argument, which must point to an uninitialized event. It returns 0 on success, and -1 on an internal error or bad arguments.

Example

```
#include <event2/event.h>
/* Watch out!  Including event_struct.h means that your code will not
 * be binary-compatible with future versions of Libevent. */
#include <event2/event_struct.h>
#include <stdlib.h>

struct event_pair {
         evutil_socket_t fd;
         struct event read_event;
         struct event write_event;
};
void readcb(evutil_socket_t, short, void *);
void writecb(evutil_socket_t, short, void *);
struct event_pair *event_pair_new(struct event_base *base, evutil_socket_t fd)
{
        struct event_pair *p = malloc(sizeof(struct event_pair));
        if (!p) return NULL;
        p->fd = fd;
        event_assign(&p->read_event, base, fd, EV_READ|EV_PERSIST, readcb, p);
        event_assign(&p->write_event, base, fd, EV_WRITE|EV_PERSIST, writecb, p);
        return p;
}
```

You can also use event_assign() to initialize stack-allocated or statically allocated events.

WARNING Never call event_assign() on an event that is already pending in an event base. Doing so can lead to extremely hard-to-diagnose errors. If the event is already initialized and pending, call event_del() on it **before** you call event_assign() on it again.

There are convenience macros you can use to event_assign() a timeout-only or a signal event:

Interface

```
#define evtimer_assign(event, base, callback, arg) \
    event_assign(event, base, -1, 0, callback, arg)
#define evsignal_assign(event, base, signum, callback, arg) \
    event_assign(event, base, signum, EV_SIGNAL|EV_PERSIST, callback, arg)
```

If you need to use event_assign() **and** retain binary compatibility with future versions of Libevent, you can ask the Libevent library to tell you at runtime how large a *struct event* should be:

Interface

```
size_t event_get_struct_event_size(void);
```

This function returns the number of bytes you need to set aside for a struct event. As before, you should only be using this function if you know that heap-allocation is actually a significant problem in your program, since it can make your code much harder to read and write.

Note that event_get_struct_event_size() may in the future give you a value *smaller* than *sizeof(struct event)*. If this happens, it means that any extra bytes at the end of *struct event* are only padding bytes reserved for use by a future version of Libevent.

Here's the same example as above, but instead of relying on the size of *struct event* from event_struct.h, we use event_get_struct_size() to use the correct size at runtime.

Example

```
#include <event2/event.h>
#include <stdlib.h>

/* When we allocate an event_pair in memory, we'll actually allocate
 * more space at the end of the structure.  We define some macros
 * to make accessing those events less error-prone. */
struct event_pair {
        evutil_socket_t fd;
};

/* Macro: yield the struct event 'offset' bytes from the start of 'p' */
#define EVENT_AT_OFFSET(p, offset) \
            ((struct event*) ( ((char*)(p)) + (offset) ))
/* Macro: yield the read event of an event_pair */
#define READEV_PTR(pair) \
            EVENT_AT_OFFSET((pair), sizeof(struct event_pair))
/* Macro: yield the write event of an event_pair */
#define WRITEEV_PTR(pair) \
            EVENT_AT_OFFSET((pair), \
                sizeof(struct event_pair)+event_get_struct_event_size())

/* Macro: yield the actual size to allocate for an event_pair */
#define EVENT_PAIR_SIZE() \
            (sizeof(struct event_pair)+2*event_get_struct_event_size())

void readcb(evutil_socket_t, short, void *);
void writecb(evutil_socket_t, short, void *);
struct event_pair *event_pair_new(struct event_base *base, evutil_socket_t fd)
{
        struct event_pair *p = malloc(EVENT_PAIR_SIZE());
        if (!p) return NULL;
        p->fd = fd;
```

```
        event_assign(READEV_PTR(p), base, fd, EV_READ|EV_PERSIST, readcb, p);
        event_assign(WRITEEV_PTR(p), base, fd, EV_WRITE|EV_PERSIST, writecb, p);
        return p;
}
```

The event_assign() function defined in <event2/event.h>. It has existed since Libevent 2.0.1-alpha. It has returned an int since 2.0.3-alpha; previously, it returned void. The event_get_struct_event_size() function was introduced in Libevent 2.0.4-alpha. The event structure itself is defined in <event2/event_struct.h>.

Making events pending and non-pending

Once you have constructed an event, it won't actually do anything until you have made it *pending* by adding it. You do this with event_add:

Interface

```
int event_add(struct event *ev, const struct timeval *tv);
```

Calling event_add on a non-pending event makes it pending in its configured base. The function returns 0 on success, and -1 on failure. If *tv* is NULL, the event is added with no timeout. Otherwise, *tv* is the size of the timeout in seconds and microseconds.

If you call event_add() on an event that is *already* pending, it will leave it pending, and reschedule it with the provided timeout. If the event is already pending, and you re-add it with the timeout NULL, event_add() will have no effect.

Note
Do not set *tv* to the time at which you want the timeout to run. If you say "tv→tv_sec = time(NULL)+10;" on 1 January 2010, your timeout will wait 40 years, not 10 seconds.

Interface

```
int event_del(struct event *ev);
```

Calling event_del on an initialized event makes it non-pending and non-active. If the event was not pending or active, there is no effect. The return value is 0 on success, -1 on failure.

Note
If you delete an event after it becomes active but before its callback has a chance to execute, the callback will not be executed.

Interface

```
int event_remove_timer(struct event *ev);
```

Finally, you can remove a pending event's timeout completely without deleting its IO or signal components. If the event had no timeout pending, event_remove_timer() has no effect. If the event had only a timeout but no IO or signal component, event_remove_timer() has the same effect as event_del(). The return value is 0 on success, -1 on failure.

These are defined in <event2/event.h>; event_add() and event_del() have existed since Libevent 0.1; event_remove_timer() was added in 2.1.2-alpha.

Events with priorities

When multiple events trigger at the same time, Libevent does not define any order with respect to when their callbacks will be executed. You can define some events as more important than others by using priorities.

As discussed in an earlier section, each event_base has one or more priority values associated with it. Before adding an event to the event_base, but after initializing it, you can set its priority.

Interface

```
int event_priority_set(struct event *event, int priority);
```

The priority of the event is a number between 0 and the number of priorities in an event_base, minus 1. The function returns 0 on success, and -1 on failure.

When multiple events of multiple priorities become active, the low-priority events are not run. Instead, Libevent runs the high priority events, then checks for events again. Only when no high-priority events are active are the low-priority events run.

Example

```
#include <event2/event.h>

void read_cb(evutil_socket_t, short, void *);
void write_cb(evutil_socket_t, short, void *);

void main_loop(evutil_socket_t fd)
{
  struct event *important, *unimportant;
  struct event_base *base;

  base = event_base_new();
  event_base_priority_init(base, 2);
  /* Now base has priority 0, and priority 1 */
  important = event_new(base, fd, EV_WRITE|EV_PERSIST, write_cb, NULL);
  unimportant = event_new(base, fd, EV_READ|EV_PERSIST, read_cb, NULL);
  event_priority_set(important, 0);
  event_priority_set(unimportant, 1);

  /* Now, whenever the fd is ready for writing, the write callback will
     happen before the read callback.  The read callback won't happen at
     all until the write callback is no longer active. */
}
```

When you do not set the priority for an event, the default is the number of queues in the event base, divided by 2.

This function is declared in <event2/event.h>. It has existed since Libevent 1.0.

Inspecting event status

Sometimes you want to tell whether an event has been added, and check what it refers to.

Interface

```
int event_pending(const struct event *ev, short what, struct timeval *tv_out);

#define event_get_signal(ev) /* ... */
evutil_socket_t event_get_fd(const struct event *ev);
struct event_base *event_get_base(const struct event *ev);
short event_get_events(const struct event *ev);
event_callback_fn event_get_callback(const struct event *ev);
void *event_get_callback_arg(const struct event *ev);
int event_get_priority(const struct event *ev);

void event_get_assignment(const struct event *event,
        struct event_base **base_out,
        evutil_socket_t *fd_out,
        short *events_out,
        event_callback_fn *callback_out,
        void **arg_out);
```

The event_pending function determines whether the given event is pending or active. If it is, and any of the flags EV_READ, EV_WRITE, EV_SIGNAL, and EV_TIMEOUT are set in the *what* argument, the function returns all of the flags that the event is currently pending or active on. If *tv_out* is provided, and EV_TIMEOUT is set in *what*, and the event is currently pending or active on a timeout, then *tv_out* is set to hold the time when the event's timeout will expire.

The event_get_fd() and event_get_signal() functions return the configured file descriptor or signal number for an event. The event_get_base() function returns its configured event_base. The event_get_events() function returns the event flags (EV_READ, EV_WRITE, etc) of the event. The event_get_callback() and event_get_callback_arg() functions return the callback function and argument pointer. The event_get_priority() function returns the event's currently assigned priority.

The event_get_assignment() function copies all of the assigned fields of the event into the provided pointers. If any of the pointers is NULL, it is ignored.

Example

```
#include <event2/event.h>
#include <stdio.h>

/* Change the callback and callback_arg of 'ev', which must not be
 * pending. */
int replace_callback(struct event *ev, event_callback_fn new_callback,
    void *new_callback_arg)
{
    struct event_base *base;
    evutil_socket_t fd;
    short events;

    int pending;

    pending = event_pending(ev, EV_READ|EV_WRITE|EV_SIGNAL|EV_TIMEOUT,
                            NULL);
    if (pending) {
        /* We want to catch this here so that we do not re-assign a
         * pending event.  That would be very very bad. */
        fprintf(stderr,
                "Error! replace_callback called on a pending event!\n");
        return -1;
    }

    event_get_assignment(ev, &base, &fd, &events,
                         NULL /* ignore old callback */ ,
                         NULL /* ignore old callback argument */);

    event_assign(ev, base, fd, events, new_callback, new_callback_arg);
    return 0;
}
```

These functions are declared in <event2/event.h>. The event_pending() function has existed since Libevent 0.1. Libevent 2.0.1-alpha introduced event_get_fd() and event_get_signal(). Libevent 2.0.2-alpha introduced event_get_base(). Libevent 2.1.2-alpha added event_get_priority(). The others were new in Libevent 2.0.4-alpha.

Finding the currently running event

For debugging or other purposes, you can get a pointer to the currently running event.

Interface

```
struct event *event_base_get_running_event(struct event_base *base);
```

Note that this function's behavior is only defined when it's called from within the provided event_base's loop. Calling it from another thread is not supported, and can cause undefined behavior.

This function is declared in <event2/event.h>. It was introduced in Libevent 2.1.1-alpha.

Configuring one-off events

If you don't need to add an event more than once, or delete it once it has been added, and it doesn't have to be persistent, you can use event_base_once().

Interface

```
int event_base_once(struct event_base *, evutil_socket_t, short,
  void (*)(evutil_socket_t, short, void *), void *, const struct timeval *);
```

This function's interface is the same as event_new(), except that it does not support EV_SIGNAL or EV_PERSIST. The scheduled event is inserted and run with the default priority. When the callback is finally done, Libevent frees the internal event structure itself. The return value is 0 on success, -1 on failure.

Events inserted with event_base_once cannot be deleted or manually activated: if you want to be able to cancel an event, create it with the regular event_new() or event_assign() interfaces.

Note also that at up to Libevent 2.0, if the event is never triggered, the internal memory used to hold it will never be freed. Starting in Libevent 2.1.2-alpha, these events *are* freed when the event_base is freed, even if they haven't activated, but still be aware: if there's some storage associated with their callback arguments, that storage won't be released unless your program has done something to track and release it.

Manually activating an event

Rarely, you may want to make an event active even though its conditions have not triggered.

Interface

```
void event_active(struct event *ev, int what, short ncalls);
```

This function makes an event *ev* become active with the flags *what* (a combination of EV_READ, EV_WRITE, and EV_TIMEOUT). The event does not need to have previously been pending, and activating it does not make it pending.

Warning: calling event_active() recursively on the same event may result in resource exhaustion. The following snippet of code is an example of how event_active can be used incorrectly.

Bad Example: making an infinite loop with event_active()

```
struct event *ev;

static void cb(int sock, short which, void *arg) {
        /* Whoops: Calling event_active on the same event unconditionally
           from within its callback means that no other events might not get
           run! */

        event_active(ev, EV_WRITE, 0);
}

int main(int argc, char **argv) {
        struct event_base *base = event_base_new();

        ev = event_new(base, -1, EV_PERSIST | EV_READ, cb, NULL);

        event_add(ev, NULL);

        event_active(ev, EV_WRITE, 0);

        event_base_loop(base, 0);

        return 0;
}
```

This creates a situation where the event loop is only executed once and calls the function "cb" forever.

Example: Alternative solution to the above problem using timers

```
struct event *ev;
struct timeval tv;

static void cb(int sock, short which, void *arg) {
   if (!evtimer_pending(ev, NULL)) {
       event_del(ev);
       evtimer_add(ev, &tv);
   }
}

int main(int argc, char **argv) {
   struct event_base *base = event_base_new();

   tv.tv_sec = 0;
   tv.tv_usec = 0;

   ev = evtimer_new(base, cb, NULL);

   evtimer_add(ev, &tv);

   event_base_loop(base, 0);

   return 0;
}
```

Example: Alternative solution to the above problem using event_config_set_max_dispatch_interval()

```
struct event *ev;

static void cb(int sock, short which, void *arg) {
        event_active(ev, EV_WRITE, 0);
}

int main(int argc, char **argv) {
        struct event_config *cfg = event_config_new();
        /* Run at most 16 callbacks before checking for other events. */
        event_config_set_max_dispatch_interval(cfg, NULL, 16, 0);
        struct event_base *base = event_base_new_with_config(cfg);
        ev = event_new(base, -1, EV_PERSIST | EV_READ, cb, NULL);

        event_add(ev, NULL);

        event_active(ev, EV_WRITE, 0);

        event_base_loop(base, 0);

        return 0;
}
```

This function is defined in <event2/event.h>. It has existed since Libevent 0.3.

Optimizing common timeouts

Current versions of Libevent use a binary heap algorithm to keep track of pending events' timeouts. A binary heap gives performance of order O(lg n) for adding and deleting each event timeout. This is optimal if you're adding events with a randomly distributed set of timeout values, but not if you have a large number of events with the same timeout.

For example, suppose you have ten thousand events, each of which should trigger its timeout five seconds after it was added. In a situation like this, you could get O(1) performance for each timeout by using a doubly-linked queue implementation.

Naturally, you wouldn't want to use a queue for all of your timeout values, since a queue is only faster for constant timeout values. If some of the timeouts are more-or-less randomly distributed, then adding one of those timeouts to a queue would take O(n) time, which would be significantly worse than a binary heap.

Libevent lets you solve this by placing some of your timeouts in queues, and others in the binary heap. To do this, you ask Libevent for a special "common timeout" timeval, which you then use to add events having that timeval. If you have a very large number of events with a single common timeout, using this optimization should improve timeout performance.

Interface

```
const struct timeval *event_base_init_common_timeout(
    struct event_base *base, const struct timeval *duration);
```

This function takes as its arguments an event_base, and the duration of the common timeout to initialize. It returns a pointer to a special struct timeval that you can use to indicate that an event should be added to an O(1) queue rather than the O(lg n) heap. This special timeval can be copied or assigned freely in your code. It will only work with the specific base you used to construct it. Do not rely on its actual contents: Libevent uses them to tell itself which queue to use.

Example

```
#include <event2/event.h>
#include <string.h>

/* We're going to create a very large number of events on a given base,
 * nearly all of which have a ten-second timeout.  If initialize_timeout
 * is called, we'll tell Libevent to add the ten-second ones to an O(1)
 * queue. */
struct timeval ten_seconds = { 10, 0 };

void initialize_timeout(struct event_base *base)
{
    struct timeval tv_in = { 10, 0 };
    const struct timeval *tv_out;
    tv_out = event_base_init_common_timeout(base, &tv_in);
    memcpy(&ten_seconds, tv_out, sizeof(struct timeval));
}

int my_event_add(struct event *ev, const struct timeval *tv)
{
    /* Note that ev must have the same event_base that we passed to
       initialize_timeout */
    if (tv && tv->tv_sec == 10 && tv->tv_usec == 0)
        return event_add(ev, &ten_seconds);
    else
        return event_add(ev, tv);
}
```

As with all optimization functions, you should avoid using the common_timeout functionality unless you're pretty sure that it matters for you.

This functionality was introduced in Libevent 2.0.4-alpha.

Telling a good event apart from cleared memory

Libevent provides functions that you can use to distinguish an initialized event from memory that has been cleared by setting it to 0 (for example, by allocating it with calloc() or clearing it with memset() or bzero()).

Interface

```
int event_initialized(const struct event *ev);

#define evsignal_initialized(ev) event_initialized(ev)
#define evtimer_initialized(ev) event_initialized(ev)
```

Warning These functions can't reliably distinguish between an initialized event and a hunk of uninitialized memory. You should not use them unless you know that the memory in question is either cleared or initialized as an event.

Generally, you shouldn't need to use these functions unless you've got a pretty specific application in mind. Events returned by event_new() are always initialized.

Example

```
#include <event2/event.h>
#include <stdlib.h>

struct reader {
    evutil_socket_t fd;
};

#define READER_ACTUAL_SIZE() \
    (sizeof(struct reader) + \
     event_get_struct_event_size())

#define READER_EVENT_PTR(r) \
    ((struct event *) (((char*)(r))+sizeof(struct reader)))

struct reader *allocate_reader(evutil_socket_t fd)
{
    struct reader *r = calloc(1, READER_ACTUAL_SIZE());
    if (r)
        r->fd = fd;
    return r;
}

void readcb(evutil_socket_t, short, void *);
int add_reader(struct reader *r, struct event_base *b)
{
    struct event *ev = READER_EVENT_PTR(r);
    if (!event_initialized(ev))
        event_assign(ev, b, r->fd, EV_READ, readcb, r);
    return event_add(ev, NULL);
}
```

The event_initialized() function has been present since Libevent 0.3.

Obsolete event manipulation functions

Pre-2.0 versions of Libevent did not have event_assign() or event_new(). Instead, you had event_set(), which associated the event with the "current" base. If you had more than one base, you needed to remember to call event_base_set() afterwards to make sure that the event was associated with the base you actually wanted to use.

Interface

```
void event_set(struct event *event, evutil_socket_t fd, short what,
        void(*callback)(evutil_socket_t, short, void *), void *arg);
int event_base_set(struct event_base *base, struct event *event);
```

The event_set() function was like event_assign(), except for its use of the current base. The event_base_set() function changes the base associated with an event.

There were variants of event_set() for dealing more conveniently with timers and signals: evtimer_set() corresponded roughly to evtimer_assign(), and evsignal_set() corresponded roughly to evsignal_assign().

Versions of Libevent before 2.0 used "signal_" as the prefix for the signal-based variants of event_set() and so on, rather than "evsignal_". (That is, they had signal_set(), signal_add(), signal_del(), signal_pending(), and signal_initialized().) Truly ancient versions of Libevent (before 0.6) used "timeout_" instead of "evtimer_". Thus, if you're doing code archeology, you might see timeout_add(), timeout_del(), timeout_initialized(), timeout_set(), timeout_pending(), and so on.

In place of the event_get_fd() and event_get_signal() functions, older versions of Libevent (before 2.0) used two macros called EVENT_FD() and EVENT_SIGNAL(). These macros inspected the event structure's contents directly and so prevented binary compatibility between versions; in 2.0 and later they are just aliases for event_get_fd() and event_get_signal().

Since versions of Libevent before 2.0 did not have locking support, it wasn't safe to call any of the functions that change an event's state with respect to a base from outside the thread running the base. These include event_add(), event_del(), event_active(), and event_base_once().

There was also an event_once() function that played the role of event_base_once(), but used the current base.

The EV_PERSIST flag did not interoperate sensibly with timeouts before Libevent 2.0. Instead resetting the timeout whenever the event was activated, the EV_PERSIST flag did nothing with the timeout.

Libevent versions before 2.0 did not support having multiple events inserted at the same time with the same fd and the same READ/WRITE. In other words, only one event at a time could be waiting for read on each fd, and only one event at a time could be waiting for write on each fd.

Helper functions and types for Libevent

The <event2/util.h> header defines many functions that you might find helpful for implementing portable applications using Libevent. Libevent uses these types and functions internally.

Basic types

evutil_socket_t

Most everywhere except Windows, a socket is an int, and the operating system hands them out in numeric order. Using the Windows socket API, however, a socket is of type SOCKET, which is really a pointer-like OS handle, and the order you receive them is undefined. We define the evutil_socket_t type to be an integer that can hold the output of socket() or accept() without risking pointer truncation on Windows.

Definition

```
#ifdef WIN32
#define evutil_socket_t intptr_t
#else
#define evutil_socket_t int
#endif
```

This type was introduced in Libevent 2.0.1-alpha.

Standard integer types

Often you will find yourself on a C system that missed out on the 21st century and therefore does not implement the standard C99 stdint.h header. For this situation, Libevent defines its own versions of the bit-width-specific integers from stdint.h:

Type	Width	Signed	Maximum	Minimum
ev_uint64_t	64	No	EV_UINT64_MAX	0
ev_int64_t	64	Yes	EV_INT64_MAX	EV_INT64_MIN
ev_uint32_t	32	No	EV_UINT32_MAX	0
ev_int32_t	32	Yes	EV_INT32_MAX	EV_INT32_MIN
ev_uint16_t	16	No	EV_UINT16_MAX	0
ev_int16_t	16	Yes	EV_INT16_MAX	EV_INT16_MIN
ev_uint8_t	8	No	EV_UINT8_MAX	0
ev_int8_t	8	Yes	EV_INT8_MAX	EV_INT8_MIN

As in the C99 standard, each type has exactly the specified width, in bits.

These types were introduced in Libevent 1.4.0-beta. The MAX/MIN constants first appeared in Libevent 2.0.4-alpha.

Miscellaneous compatibility types

The ev_ssize_t type is defined to ssize_t (signed size_t) on platforms that have one, and to a reasonable default on platforms that don't. The largest possible value of ev_ssize_t is EV_SSIZE_MAX; the smallest is EV_SSIZE_MIN. (The largest possible value for size_t is EV_SIZE_MAX, in case your platform doesn't define a SIZE_MAX for you.)

The ev_off_t type is used to represent offset into a file or a chunk of memory. It is defined to off_t on platforms with a reasonable off_t definition, and to ev_int64_t on Windows.

Some implementations of the sockets API provide a length type, socklen_t, and some do not. The ev_socklen_t is defined to this type where it exists, and a reasonable default otherwise.

The ev_intptr_t type is a signed integer that is large enough to hold a pointer without loss of bits. The ev_uintptr_t type is an unsigned integer large enough to hold a pointer without loss of bits.

The ev_ssize_t type was added in Libevent 2.0.2-alpha. The ev_socklen_t type was new in Libevent 2.0.3-alpha. The ev_intptr_t and ev_uintptr_t types, and the EV_SSIZE_MAX/MIN macros, were added in Libevent 2.0.4-alpha. The ev_off_t type first appeared in Libevent 2.0.9-rc.

Timer portability functions

Not every platform defines the standard timeval manipulation functions, so we provide our own implementations.

Interface

```
#define evutil_timeradd(tvp, uvp, vvp) /* ... */
#define evutil_timersub(tvp, uvp, vvp) /* ... */
```

These macros add or subtract (respectively) their first two arguments, and stores the result in the third.

Interface

```
#define evutil_timerclear(tvp) /* ... */
#define evutil_timerisset(tvp) /* ... */
```

Clearing a timeval sets its value to zero. Checking whether it is set returns true if it is nonzero and false otherwise.

Interface

```
#define evutil_timercmp(tvp, uvp, cmp)
```

The evutil_timercmp macro compares two timevals, and yields true if they are in the relationship specified by the relational operator *cmp*. For example, *evutil_timercmp(t1, t2, ⇐)* means, "Is t1 ⇐ t2?" Note that unlike some operating systems' versions, Libevent's timercmp supports all the C relational operations (that is, <, >, ==, !=, ⇐, and >=).

Interface

```
int evutil_gettimeofday(struct timeval *tv, struct timezone *tz);
```

The evutil_gettimeofday function sets *tv* to the current time. The tz argument is unused.

Example

```
struct timeval tv1, tv2, tv3;

/* Set tv1 = 5.5 seconds */
tv1.tv_sec = 5; tv1.tv_usec = 500*1000;

/* Set tv2 = now */
evutil_gettimeofday(&tv2, NULL);

/* Set tv3 = 5.5 seconds in the future */
evutil_timeradd(&tv1, &tv2, &tv3);

/* all 3 should print true */
if (evutil_timercmp(&tv1, &tv1, ==))  /* == "If tv1 == tv1" */
   puts("5.5 sec == 5.5 sec");
if (evutil_timercmp(&tv3, &tv2, >=))  /* == "If tv3 >= tv2" */
   puts("The future is after the present.");
if (evutil_timercmp(&tv1, &tv2, <))   /* == "If tv1 < tv2" */
   puts("It is no longer the past.");
```

These functions were introduced in Libevent 1.4.0-beta, except for evutil_gettimeofday(), which was introduced in Libevent 2.0.

Note
It wasn't safe to use ⇐ or >= with timercmp before Libevent 1.4.4.

Socket API compatibility

This section exists because, for historical reasons, Windows has never really implemented the Berkeley sockets API in a nice compatible (and nicely compatible) way. Here are some functions you can use in order to pretend that it has.

Interface

```
int evutil_closesocket(evutil_socket_t s);

#define EVUTIL_CLOSESOCKET(s) evutil_closesocket(s)
```

This function closes a socket. On Unix, it's an alias for close(); on Windows, it calls closesocket(). (You can't use close() on sockets on Windows, and nobody else defines a closesocket().)

The evutil_closesocket function was introduced in Libevent 2.0.5-alpha. Before then, you needed to call the EVUTIL_CLOSESOCKET macro.

Interface

```
#define EVUTIL_SOCKET_ERROR()
#define EVUTIL_SET_SOCKET_ERROR(errcode)
#define evutil_socket_geterror(sock)
#define evutil_socket_error_to_string(errcode)
```

These macros access and manipulate socket error codes. EVUTIL_SOCKET_ERROR() returns the global error code for the last socket operation from this thread, and evutil_socket_geterror() does so for a particular socket. (Both are errno on Unix-like systems.) EVUTIL_SET_SOCKET_ERROR() changes the current socket error code (like setting errno on Unix), and evutil_socket_error_to_string() returns a string representation of a given socket error code (like strerror() on Unix).

(We need these functions because Windows doesn't use errno for errors from socket functions, but instead uses WSAGetLastError().)

Note that the Windows socket errors are not the same as the standard-C errors you would see in errno; watch out.

Interface

```
int evutil_make_socket_nonblocking(evutil_socket_t sock);
```

Even the call you need to do nonblocking IO on a socket is not portable to Windows. The evutil_make_socket_nonblocking() function takes a new socket (from socket() or accept()) and turns it into a nonblocking socket. (It sets O_NONBLOCK on Unix and FIONBIO on Windows.)

Interface

```
int evutil_make_listen_socket_reuseable(evutil_socket_t sock);
```

This function makes sure that the address used by a listener socket will be available to another socket immediately after the socket is closed. (It sets SO_REUSEADDR on Unix and does nothing on Windows. You don't want to use SO_REUSEADDR on Windows; it means something different there.)

Interface

```
int evutil_make_socket_closeonexec(evutil_socket_t sock);
```

This call tells the operating system that this socket should be closed if we ever call exec(). It sets the FD_CLOEXEC flag on Unix, and does nothing on Windows.

Interface

```
int evutil_socketpair(int family, int type, int protocol,
        evutil_socket_t sv[2]);
```

This function behaves as the Unix socketpair() call: it makes two sockets that are connected with each other and can be used with ordinary socket IO calls. It stores the two sockets in sv[0] and sv[1], and returns 0 for success and -1 for failure.

On Windows, this only supports family AF_INET, type SOCK_STREAM, and protocol 0. Note that this can fail on some Windows hosts where firewall software has cleverly firewalled 127.0.0.1 to keep the host from talking to itself.

These functions were introduced in Libevent 1.4.0-beta, except for evutil_make_socket_closeonexec(), which was new in Libevent 2.0.4-alpha.

Portable string manipulation functions

Interface

```
ev_int64_t evutil_strtoll(const char *s, char **endptr, int base);
```

This function behaves as strtol, but handles 64-bit integers. On some platforms, it only supports Base 10.

Interface

```
int evutil_snprintf(char *buf, size_t buflen, const char *format, ...);
int evutil_vsnprintf(char *buf, size_t buflen, const char *format, va_list ap);
```

These snprintf-replacement functions behave as the standard snprintf and vsnprintf interfaces. They return the number of bytes that would have been written into the buffer had it been long enough, not counting the terminating NUL byte. (This behavior conforms to the C99 snprintf() standard, and is in contrast to the Windows _snprintf(), which returns a negative number if the string would not fit in the buffer.)

The evutil_strtoll() function has been in Libevent since 1.4.2-rc. These other functions first appeared in version 1.4.5.

Locale-independent string manipulation functions

Sometimes, when implementing ASCII-based protocols, you want to manipulate strings according to ASCII's notion of character type, regardless of your current locale. Libevent provides a few functions to help with this:

Interface

```
int evutil_ascii_strcasecmp(const char *str1, const char *str2);
int evutil_ascii_strncasecmp(const char *str1, const char *str2, size_t n);
```

These functions behave as strcasecmp() and strncasecmp(), except that they always compare using the ASCII character set, regardless of the current locale. The evutil_ascii_str[n]casecmp() functions were first exposed in Libevent 2.0.3-alpha.

IPv6 helper and portability functions

Interface

```
const char *evutil_inet_ntop(int af, const void *src, char *dst, size_t len);
int evutil_inet_pton(int af, const char *src, void *dst);
```

These functions behave as the standard inet_ntop() and inet_pton() functions for parsing and formatting IPv4 and IPv6 addresses, as specified in RFC3493. That is, to format an IPv4 address, you call evutil_inet_ntop() with *af* set to AF_INET, *src* pointing to a struct in_addr, and *dst* pointing to a character buffer of size *len*. For an IPv6 address, *af* is AF_INET6 and *src* is a struct in6_addr. To parse an IPv4 address, call evutil_inet_pton() with *af* set to AF_INET or AF_INET6, the string to parse in *src*, and *dst* pointing to an in_addr or an in_addr6 as appropriate.

The return value from evutil_inet_ntop() is NULL on failure and otherwise points to dst. The return value from evutil_inet_pton() is 0 on success and -1 on failure.

Interface

```
int evutil_parse_sockaddr_port(const char *str, struct sockaddr *out,
    int *outlen);
```

This function parses an address from *str* and writes the result to *out*. The *outlen* argument must point to an integer holding the number of bytes available in *out*; it is altered to hold the number of bytes actually used. This function returns 0 on success and -1 on failure. It recognizes the following address formats:

- [ipv6]:port (as in "[ffff::]:80")
- ipv6 (as in "ffff::")
- [ipv6] (as in "[ffff::]")
- ipv4:port (as in "1.2.3.4:80")
- ipv4 (as in "1.2.3.4")

If no port is given, the port in the resulting sockaddr is set to 0.

Interface

```
int evutil_sockaddr_cmp(const struct sockaddr *sa1,
    const struct sockaddr *sa2, int include_port);
```

The evutil_sockaddr_cmp() function compares two addresses, and returns negative if sa1 precedes sa2, 0 if they are equal, and positive if sa2 precedes sa1. It works for AF_INET and AF_INET6 addresses, and returns undefined output for other addresses. It's guaranteed to give a total order for these addresses, but the ordering may change between Libevent versions.

If the *include_port* argument is false, then two sockaddrs are treated as equal if they differ only in their port. Otherwise, sockaddrs with different ports are treated as unequal.

These functions were introduced in Libevent 2.0.1-alpha, except for evutil_sockaddr_cmp(), which introduced in 2.0.3-alpha.

Structure macro portability functions

Interface

```
#define evutil_offsetof(type, field) /* ... */
```

As the standard offsetof macro, this macro yields the number of bytes from the start of *type* at which *field* occurs.

This macro was introduced in Libevent 2.0.1-alpha. It was buggy in every version before Libevent 2.0.3-alpha.

Secure random number generator

Many applications (including evdns) need a source of hard-to-predict random numbers for their security.

Interface

```
void evutil_secure_rng_get_bytes(void *buf, size_t n);
```

This function fills n-byte buffer at *buf* with *n* bytes of random data.

If your platform provides the arc4random() function, Libevent uses that. Otherwise, it uses its own implementation of arc4random(), seeded by your operating system's entropy pool (CryptGenRandom on Windows, /dev/urandom everywhere else).

Interface

```
int evutil_secure_rng_init(void);
void evutil_secure_rng_add_bytes(const char *dat, size_t datlen);
```

You do not need to manually initialize the secure random number generator, but if you want to make sure it is successfully initialized, you can do so by calling evutil_secure_rng_init(). It seeds the RNG (if it was not already seeded) and returns 0 on success. If it returns -1, Libevent wasn't able to find a good source of entropy on your OS, and you can't use the RNG safely without initializing it yourself.

If you are running in an environment where your program is likely to drop privileges (for example, by running chroot()), you should call evutil_secure_rng_init() before you do so.

You can add more random bytes to the entropy pool yourself by calling evutil_secure_rng_add_bytes(); this shouldn't be necessary in typical use.

These functions are new in Libevent 2.0.4-alpha.

Bufferevents: concepts and basics

Most of the time, an application wants to perform some amount of data buffering in addition to just responding to events. When we want to write data, for example, the usual pattern runs something like:

- Decide that we want to write some data to a connection; put that data in a buffer.
- Wait for the connection to become writable

- Write as much of the data as we can
- Remember how much we wrote, and if we still have more data to write, wait for the connection to become writable again.

This buffered IO pattern is common enough that Libevent provides a generic mechanism for it. A "bufferevent" consists of an underlying transport (like a socket), a read buffer, and a write buffer. Instead of regular events, which give callbacks when the underlying transport is ready to be read or written, a bufferevent invokes its user-supplied callbacks when it has read or written enough data.

There are multiple types of bufferevent that all share a common interface. As of this writing, the following types exist:

socket-based bufferevents
A bufferevent that sends and receives data from an underlying stream socket, using the event_* interface as its backend.

asynchronous-IO bufferevents
A bufferevent that uses the Windows IOCP interface to send and receive data to an underlying stream socket. (Windows only; experimental.)

filtering bufferevents
A bufferevent that processes incoming and outgoing data before passing it to an underlying bufferevent object—for example, to compress or translate data.

paired bufferevents
Two bufferevents that transmit data to one another.

NOTE As of Libevent 2.0.2-alpha, the bufferevents interfaces here are still not fully orthogonal across all bufferevent types. In other words, not every interface described below will work on all bufferevent types. The Libevent developers intend to correct this in future versions.

NOTE ALSO Bufferevents currently only work for stream-oriented protocols like TCP. There may in the future be support for datagram-oriented protocols like UDP.

All of the functions and types in this section are declared in event2/bufferevent.h. Functions specifically related to evbuffers are declared in event2/buffer.h; see the next chapter for information on those.

Bufferevents and evbuffers

Every bufferevent has an input buffer and an output buffer. These are of type "struct evbuffer". When you have data to write on a bufferevent, you add it to the output buffer; when a bufferevent has data for you to read, you drain it from the input buffer.

The evbuffer interface supports many operations; we discuss them in a later section.

Callbacks and watermarks

Every bufferevent has two data-related callbacks: a read callback and a write callback. By default, the read callback is called whenever any data is read from the underlying transport, and the write callback is called whenever enough data from the output buffer is emptied to the underlying transport. You can override the behavior of these functions by adjusting the read and write "watermarks" of the bufferevent.

Every bufferevent has four watermarks:

Read low-water mark
Whenever a read occurs that leaves the bufferevent's input buffer at this level or higher, the bufferevent's read callback is invoked. Defaults to 0, so that every read results in the read callback being invoked.

Read high-water mark
If the bufferevent's input buffer ever gets to this level, the bufferevent stops reading until enough data is drained from the input buffer to take us below it again. Defaults to unlimited, so that we never stop reading because of the size of the input buffer.

Write low-water mark
Whenever a write occurs that takes us to this level or below, we invoke the write callback. Defaults to 0, so that a write callback is not invoked unless the output buffer is emptied.

Write high-water mark
Not used by a bufferevent directly, this watermark can have special meaning when a bufferevent is used as the underlying transport of another bufferevent. See notes on filtering bufferevents below.

A bufferevent also has an "error" or "event" callback that gets invoked to tell the application about non-data-oriented events, like when a connection is closed or an error occurs. The following event flags are defined:

BEV_EVENT_READING
An event occured during a read operation on the bufferevent. See the other flags for which event it was.

BEV_EVENT_WRITING
An event occured during a write operation on the bufferevent. See the other flags for which event it was.

BEV_EVENT_ERROR
An error occurred during a bufferevent operation. For more information on what the error was, call EVUTIL_SOCKET_ERROR().

BEV_EVENT_TIMEOUT
A timeout expired on the bufferevent.

BEV_EVENT_EOF
We got an end-of-file indication on the bufferevent.

BEV_EVENT_CONNECTED
We finished a requested connection on the bufferevent.

(The above event names are new in Libevent 2.0.2-alpha.)

Deferred callbacks

By default, a bufferevent callbacks are executed *immediately* when the corresponding condition happens. (This is true of evbuffer callbacks too; we'll get to those later.) This immediate invocation can make trouble when dependencies get complex. For example, suppose that there is a callback that moves data into evbuffer A when it grows empty, and another callback that processes data out of evbuffer A when it grows full. Since these calls are all happening on the stack, you might risk a stack overflow if the dependency grows nasty enough.

To solve this, you can tell a bufferevent (or an evbuffer) that its callbacks should be *deferred*. When the conditions are met for a deferred callback, rather than invoking it immediately, it is queued as part of the event_loop() call, and invoked after the regular events' callbacks.

(Deferred callbacks were introduced in Libevent 2.0.1-alpha.)

Option flags for bufferevents

You can use one or more flags when creating a bufferevent to alter its behavior. Recognized flags are:

BEV_OPT_CLOSE_ON_FREE
When the bufferevent is freed, close the underlying transport. This will close an underlying socket, free an underlying bufferevent, etc.

BEV_OPT_THREADSAFE
Automatically allocate locks for the bufferevent, so that it's safe to use from multiple threads.

BEV_OPT_DEFER_CALLBACKS
When this flag is set, the bufferevent defers all of its callbacks, as described above.

BEV_OPT_UNLOCK_CALLBACKS
By default, when the bufferevent is set up to be threadsafe, the bufferevent's locks are held whenever the any user-provided callback is invoked. Setting this option makes Libevent release the bufferevent's lock when it's invoking your callbacks.

(Libevent 2.0.5-beta introduced BEV_OPT_UNLOCK_CALLBACKS. The other options above were new in Libevent 2.0.1-alpha.)

Working with socket-based bufferevents

The simplest bufferevents to work with is the socket-based type. A socket-based bufferevent uses Libevent's underlying event mechanism to detect when an underlying network socket is ready for read and/or write operations, and uses underlying network calls (like readv, writev, WSASend, or WSARecv) to transmit and receive data.

Creating a socket-based bufferevent

You can create a socket-based bufferevent using bufferevent_socket_new():

Interface

```
struct bufferevent *bufferevent_socket_new(
    struct event_base *base,
    evutil_socket_t fd,
    enum bufferevent_options options);
```

The *base* is an event_base, and *options* is a bitmask of bufferevent options (BEV_OPT_CLOSE_ON_FREE, etc). The *fd* argument is an optional file descriptor for a socket. You can set fd to -1 if you want to set the file descriptor later.

Tip
[Make sure that the socket you provide to bufferevent_socket_new is in non-blocking mode. Libevent provides the convenience method evutil_make_socket_nonblocking for this.]

This function returns a bufferevent on success, and NULL on failure.

The bufferevent_socket_new() function was introduced in Libevent 2.0.1-alpha.

Launching connections on socket-based bufferevents

If the bufferevent's socket is not yet connected, you can launch a new connection.

Interface

```
int bufferevent_socket_connect(struct bufferevent *bev,
    struct sockaddr *address, int addrlen);
```

The address and addrlen arguments are as for the standard call connect(). If the bufferevent does not already have a socket set, calling this function allocates a new stream socket for it, and makes it nonblocking.

If the bufferevent **does** have a socket already, calling bufferevent_socket_connect() tells Libevent that the socket is not connected, and no reads or writes should be done on the socket until the connect operation has succeeded.

It is okay to add data to the output buffer before the connect is done.

This function returns 0 if the connect was successfully launched, and -1 if an error occurred.

Example

```
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <sys/socket.h>
#include <string.h>

void eventcb(struct bufferevent *bev, short events, void *ptr)
{
    if (events & BEV_EVENT_CONNECTED) {
         /* We're connected to 127.0.0.1:8080.   Ordinarily we'd do
            something here, like start reading or writing. */
    } else if (events & BEV_EVENT_ERROR) {
         /* An error occured while connecting. */
    }
}

int main_loop(void)
{
    struct event_base *base;
    struct bufferevent *bev;
    struct sockaddr_in sin;

    base = event_base_new();

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(0x7f000001); /* 127.0.0.1 */
    sin.sin_port = htons(8080); /* Port 8080 */

    bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE);

    bufferevent_setcb(bev, NULL, NULL, eventcb, NULL);

    if (bufferevent_socket_connect(bev,
         (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        /* Error starting connection */
        bufferevent_free(bev);
        return -1;
    }

    event_base_dispatch(base);
    return 0;
}
```

The bufferevent_socket_connect() function was introduced in Libevent-2.0.2-alpha. Before then, you had to manually call connect() on your socket yourself, and when the connection was complete, the bufferevent would report it as a write.

Note that you only get a BEV_EVENT_CONNECTED event if you launch the connect() attempt using bufferevent_socket_connect(). If you call connect() on your own, the connection gets reported as a write.

If you want to call connect() yourself, but still get receive a BEV_EVENT_CONNECTED event when the connection succeeds, call bufferevent_socket_connect(bev, NULL, 0) after connect() returns -1 with errno equal to EAGAIN or EINPROGRESS.

This function was introduced in Libevent 2.0.2-alpha.

Launching connections by hostname

Quite often, you'd like to combine resolving a hostname and connecting to it into a single operation. There's an interface for that:

Interface

```
int bufferevent_socket_connect_hostname(struct bufferevent *bev,
```

```
    struct evdns_base *dns_base, int family, const char *hostname,
    int port);
int bufferevent_socket_get_dns_error(struct bufferevent *bev);
```

This function resolves the DNS name *hostname*, looking for addresses of type *family*. (Allowable family types are AF_INET, AF_INET6, and AF_UNSPEC.) If the name resolution fails, it invokes the event callback with an error event. If it succeeds, it launches a connection attempt just as bufferevent_connect would.

The dns_base argument is optional. If it is NULL, then Libevent blocks while waiting for the name lookup to finish, which usually isn't what you want. If it is provided, then Libevent uses it to look up the hostname asynchronously. See chapter R9 for more info on DNS.

As with bufferevent_socket_connect(), this function tells Libevent that any existing socket on the bufferevent is not connected, and no reads or writes should be done on the socket until the resolve is finished and the connect operation has succeeded.

If an error occurs, it might be a DNS hostname lookup error. You can find out what the most recent error was by calling bufferevent_socket_get_dns_error(). If the returned error code is 0, no DNS error was detected.

Example: Trivial HTTP v0 client.

```
/* Don't actually copy this code: it is a poor way to implement an
   HTTP client.  Have a look at evhttp instead.
*/
#include <event2/dns.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/util.h>
#include <event2/event.h>

#include <stdio.h>

void readcb(struct bufferevent *bev, void *ptr)
{
    char buf[1024];
    int n;
    struct evbuffer *input = bufferevent_get_input(bev);
    while ((n = evbuffer_remove(input, buf, sizeof(buf))) > 0) {
        fwrite(buf, 1, n, stdout);
    }
}

void eventcb(struct bufferevent *bev, short events, void *ptr)
{
    if (events & BEV_EVENT_CONNECTED) {
         printf("Connect okay.\n");
    } else if (events & (BEV_EVENT_ERROR|BEV_EVENT_EOF)) {
         struct event_base *base = ptr;
         if (events & BEV_EVENT_ERROR) {
                int err = bufferevent_socket_get_dns_error(bev);
                if (err)
                        printf("DNS error: %s\n", evutil_gai_strerror(err));
         }
         printf("Closing\n");
         bufferevent_free(bev);
         event_base_loopexit(base, NULL);
    }
}

int main(int argc, char **argv)
{
    struct event_base *base;
    struct evdns_base *dns_base;
    struct bufferevent *bev;
```

```
    if (argc != 3) {
        printf("Trivial HTTP 0.x client\n"
               "Syntax: %s [hostname] [resource]\n"
               "Example: %s www.google.com /\n",argv[0],argv[0]);
        return 1;
    }

    base = event_base_new();
    dns_base = evdns_base_new(base, 1);

    bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE);
    bufferevent_setcb(bev, readcb, NULL, eventcb, base);
    bufferevent_enable(bev, EV_READ|EV_WRITE);
    evbuffer_add_printf(bufferevent_get_output(bev), "GET %s\r\n", argv[2]);
    bufferevent_socket_connect_hostname(
        bev, dns_base, AF_UNSPEC, argv[1], 80);
    event_base_dispatch(base);
    return 0;
}
```

The bufferevent_socket_connect_hostname() function was new in Libevent 2.0.3-alpha; bufferevent_socket_get_dns_error() was new in 2.0.5-beta.

Generic bufferevent operations

The functions in this section work with multiple bufferevent implementations.

Freeing a bufferevent

Interface

```
void bufferevent_free(struct bufferevent *bev);
```

This function frees a bufferevent. Bufferevents are internally reference-counted, so if the bufferevent has pending deferred callbacks when you free it, it won't be deleted until the callbacks are done.

The bufferevent_free() function does, however, try to free the bufferevent as soon as possible. If there is pending data to write on the bufferevent, it probably won't be flushed before the bufferevent is freed.

If the BEV_OPT_CLOSE_ON_FREE flag was set, and this bufferevent has a socket or underlying bufferevent associated with it as its transport, that transport is closed when you free the bufferevent.

This function was introduced in Libevent 0.8.

Manipulating callbacks, watermarks, and enabled operations

Interface

```
typedef void (*bufferevent_data_cb)(struct bufferevent *bev, void *ctx);
typedef void (*bufferevent_event_cb)(struct bufferevent *bev,
    short events, void *ctx);

void bufferevent_setcb(struct bufferevent *bufev,
    bufferevent_data_cb readcb, bufferevent_data_cb writecb,
    bufferevent_event_cb eventcb, void *cbarg);

void bufferevent_getcb(struct bufferevent *bufev,
    bufferevent_data_cb *readcb_ptr,
```

```
    bufferevent_data_cb *writecb_ptr,
    bufferevent_event_cb *eventcb_ptr,
    void **cbarg_ptr);
```

The bufferevent_setcb() function changes one or more of the callbacks of a bufferevent. The readcb, writecb, and eventcb functions are called (respectively) when enough data is read, when enough data is written, or when an event occurs. The first argument of each is the bufferevent that has had the event happen. The last argument is the value provided by the user in the *cbarg* parameter of bufferevent_callcb(): You can use this to pass data to your callbacks. The *events* argument of the event callback is a bitmask of event flags: see "callbacks and watermarks" above.

You can disable a callback by passing NULL instead of the callback function. Note all the callback functions on a bufferevent share a single *cbarg* value, so changing it will affect all of them.

You can retrieve the currently set callbacks for a bufferevent by passing pointers to bufferevent_getcb(), which sets *readcb_ptr to the current read callback, *writecb_ptr to the current write callback, *eventcb_ptr to the current event callback, and *cbarg_ptr to the current callback argument field. Any of these pointers set to NULL will be ignored.

The bufferevent_setcb() function was introduced in Libevent 1.4.4. The type names "bufferevent_data_cb" and "bufferevent_event_cb" were new in Libevent 2.0.2-alpha. The bufferevent_getcb() function was added in 2.1.1-alpha.

Interface

```
void bufferevent_enable(struct bufferevent *bufev, short events);
void bufferevent_disable(struct bufferevent *bufev, short events);

short bufferevent_get_enabled(struct bufferevent *bufev);
```

You can enable or disable the events EV_READ, EV_WRITE, or EV_READ|EV_WRITE on a bufferevent. When reading or writing is not enabled, the bufferevent will not try to read or write data.

There is no need to disable writing when the output buffer is empty: the bufferevent automatically stops writing, and restarts again when there is data to write.

Similarly, there is no need to disable reading when the input buffer is up to its high-water mark: the bufferevent automatically stops reading, and restarts again when there is space to read.

By default, a newly created bufferevent has writing enabled, but not reading.

You can call bufferevent_get_enabled() to see which events are currently enabled on the bufferevent.

These functions were introduced in Libevent 0.8, except for bufferevent_get_enabled(), which was introduced in version 2.0.3-alpha.

Interface

```
void bufferevent_setwatermark(struct bufferevent *bufev, short events,
    size_t lowmark, size_t highmark);
```

The bufferevent_setwatermark() function adjusts the read watermarks, the write watermarks, or both, of a single bufferevent. (If EV_READ is set in the events field, the read watermarks are adjusted. If EV_WRITE is set in the events field, the write watermarks are adjusted.)

A high-water mark of 0 is equivalent to "unlimited".

This function was first exposed in Libevent 1.4.4.

Example

```
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/util.h>

#include <stdlib.h>
#include <errno.h>
#include <string.h>
```

```
struct info {
    const char *name;
    size_t total_drained;
};

void read_callback(struct bufferevent *bev, void *ctx)
{
    struct info *inf = ctx;
    struct evbuffer *input = bufferevent_get_input(bev);
    size_t len = evbuffer_get_length(input);
    if (len) {
        inf->total_drained += len;
        evbuffer_drain(input, len);
        printf("Drained %lu bytes from %s\n",
             (unsigned long) len, inf->name);
    }
}

void event_callback(struct bufferevent *bev, short events, void *ctx)
{
    struct info *inf = ctx;
    struct evbuffer *input = bufferevent_get_input(bev);
    int finished = 0;

    if (events & BEV_EVENT_EOF) {
        size_t len = evbuffer_get_length(input);
        printf("Got a close from %s.  We drained %lu bytes from it, "
            "and have %lu left.\n", inf->name,
            (unsigned long)inf->total_drained, (unsigned long)len);
        finished = 1;
    }
    if (events & BEV_EVENT_ERROR) {
        printf("Got an error from %s: %s\n",
            inf->name, evutil_socket_error_to_string(EVUTIL_SOCKET_ERROR()));
        finished = 1;
    }
    if (finished) {
        free(ctx);
        bufferevent_free(bev);
    }
}

struct bufferevent *setup_bufferevent(void)
{
    struct bufferevent *b1 = NULL;
    struct info *info1;

    info1 = malloc(sizeof(struct info));
    info1->name = "buffer 1";
    info1->total_drained = 0;

    /* ... Here we should set up the bufferevent and make sure it gets
       connected... */

    /* Trigger the read callback only whenever there is at least 128 bytes
       of data in the buffer. */
    bufferevent_setwatermark(b1, EV_READ, 128, 0);

    bufferevent_setcb(b1, read_callback, NULL, event_callback, info1);

    bufferevent_enable(b1, EV_READ); /* Start reading. */
```

```
    return b1;
}
```

Manipulating data in a bufferevent

Reading and writing data from the network does you no good if you can't look at it. Bufferevents give you these methods to give them data to write, and to get the data to read:

Interface

```
struct evbuffer *bufferevent_get_input(struct bufferevent *bufev);
struct evbuffer *bufferevent_get_output(struct bufferevent *bufev);
```

These two functions are very powerful fundamental: they return the input and output buffers respectively. For full information on all the operations you can perform on an evbuffer type, see the next chapter.

Note that the application may only remove (not add) data on the input buffer, and may only add (not remove) data from the output buffer.

If writing on the bufferevent was stalled because of too little data (or if reading was stalled because of too much), then adding data to the output buffer (or removing data from the input buffer) will automatically restart it.

These functions were introduced in Libevent 2.0.1-alpha.

Interface

```
int bufferevent_write(struct bufferevent *bufev,
    const void *data, size_t size);
int bufferevent_write_buffer(struct bufferevent *bufev,
    struct evbuffer *buf);
```

These functions add data to a bufferevent's output buffer. Calling bufferevent_write() adds *size* bytes from the memory at *data* to the end of the output buffer. Calling bufferevent_write_buffer() removes the entire contents of *buf* and puts them at the end of the output buffer. Both return 0 if successful, or -1 if an error occurred.

These functions have existed since Libevent 0.8.

Interface

```
size_t bufferevent_read(struct bufferevent *bufev, void *data, size_t size);
int bufferevent_read_buffer(struct bufferevent *bufev,
    struct evbuffer *buf);
```

These functions remove data from a bufferevent's input buffer. The bufferevent_read() function removes up to *size* bytes from the input buffer, storing them into the memory at *data*. It returns the number of bytes actually removed. The bufferevent_read_buffer() function drains the entire contents of the input buffer and places them into *buf*; it returns 0 on success and -1 on failure.

Note that with bufferevent_read(), the memory chunk at *data* must actually have enough space to hold *size* bytes of data.

The bufferevent_read() function has existed since Libevent 0.8; bufferevent_read_buffer() was introduced in Libevent 2.0.1-alpha.

Example

```
#include <event2/bufferevent.h>
#include <event2/buffer.h>

#include <ctype.h>

void
read_callback_uppercase(struct bufferevent *bev, void *ctx)
{
        /* This callback removes the data from bev's input buffer 128
```

```
           bytes at a time, uppercases it, and starts sending it
           back.

           (Watch out!  In practice, you shouldn't use toupper to implement
           a network protocol, unless you know for a fact that the current
           locale is the one you want to be using.)
         */

        char tmp[128];
        size_t n;
        int i;
        while (1) {
                n = bufferevent_read(bev, tmp, sizeof(tmp));
                if (n <= 0)
                        break; /* No more data. */
                for (i=0; i<n; ++i)
                        tmp[i] = toupper(tmp[i]);
                bufferevent_write(bev, tmp, n);
        }
}

struct proxy_info {
        struct bufferevent *other_bev;
};
void
read_callback_proxy(struct bufferevent *bev, void *ctx)
{
        /* You might use a function like this if you're implementing
           a simple proxy: it will take data from one connection (on
           bev), and write it to another, copying as little as
           possible. */
        struct proxy_info *inf = ctx;

        bufferevent_read_buffer(bev,
            bufferevent_get_output(inf->other_bev));
}

struct count {
        unsigned long last_fib[2];
};

void
write_callback_fibonacci(struct bufferevent *bev, void *ctx)
{
        /* Here's a callback that adds some Fibonacci numbers to the
           output buffer of bev.  It stops once we have added 1k of
           data; once this data is drained, we'll add more. */
        struct count *c = ctx;

        struct evbuffer *tmp = evbuffer_new();
        while (evbuffer_get_length(tmp) < 1024) {
                unsigned long next = c->last_fib[0] + c->last_fib[1];
                c->last_fib[0] = c->last_fib[1];
                c->last_fib[1] = next;

                evbuffer_add_printf(tmp, "%lu", next);
        }

        /* Now we add the whole contents of tmp to bev. */
        bufferevent_write_buffer(bev, tmp);

        /* We don't need tmp any longer. */
```

```
        evbuffer_free(tmp);
}
```

Read- and write timeouts

As with other events, you can have a timeout get invoked if a certain amount of time passes without any data having been successfully written or read by a bufferevent.

Interface

```
void bufferevent_set_timeouts(struct bufferevent *bufev,
    const struct timeval *timeout_read, const struct timeval *timeout_write);
```

Setting a timeout to NULL is supposed to remove it; however before Libevent 2.1.2-alpha this wouldn't work with all event types. (As a workaround for older versions, you can try setting the timeout to a multi-day interval and/or having your eventcb function ignore BEV_TIMEOUT events when you don't want them.)

The read timeout will trigger if the bufferevent waits at least *timeout_read* seconds while trying to read read. The write timeout will trigger if the bufferevent waits at least *timeout_write* seconds while trying to write data.

Note that the timeouts only count when the bufferevent would like to read or write. In other words, the read timeout is not enabled if reading is disabled on the bufferevent, or if the input buffer is full (at its high-water mark). Similarly, the write timeout is not enabled if if writing is disabled, or if there is no data to write.

When a read or write timeout occurs, the corresponding read or write operation becomes disabled on the bufferevent. The event callback is then invoked with either BEV_EVENT_TIMEOUT|BEV_EVENT_READING or BEV_EVENT_TIMEOUT|BEV_EVENT_

This functions has existed since Libevent 2.0.1-alpha. It didn't behave consistently across bufferevent types until Libevent 2.0.4-alpha.

Initiating a flush on a bufferevent

Interface

```
int bufferevent_flush(struct bufferevent *bufev,
    short iotype, enum bufferevent_flush_mode state);
```

Flushing a bufferevent tells the bufferevent to force as many bytes as possible to be read to or written from the underlying transport, ignoring other restrictions that might otherwise keep them from being written. Its detailed function depends on the type of the bufferevent.

The iotype argument should be EV_READ, EV_WRITE, or EV_READ|EV_WRITE to indicate whether bytes being read, written, or both should be processed. The state argument may be one of BEV_NORMAL, BEV_FLUSH, or BEV_FINISHED. BEV_FINISHED indicates that the other side should be told that no more data will be sent; the distinction between BEV_NORMAL and BEV_FLUSH depends on the type of the bufferevent.

The bufferevent_flush() function returns -1 on failure, 0 if no data was flushed, or 1 if some data was flushed.

Currently (as of Libevent 2.0.5-beta), bufferevent_flush() is only implemented for some bufferevent types. In particular, socket-based bufferevents don't have it.

Type-specific bufferevent functions

These bufferevent functions are not supported on all bufferevent types.

Interface

```
int bufferevent_priority_set(struct bufferevent *bufev, int pri);
int bufferevent_get_priority(struct bufferevent *bufev);
```

This function adjusts the priority of the events used to implement *bufev* to *pri*. See event_priority_set() for more information on priorities.

This function returns 0 on success, and -1 on failure. It works on socket-based bufferevents only.

The bufferevent_priority_set() function was introduced in Libevent 1.0; bufferevent_get_priority() didn't appear until Libevent 2.1.2-alpha.

Interface

```
int bufferevent_setfd(struct bufferevent *bufev, evutil_socket_t fd);
evutil_socket_t bufferevent_getfd(struct bufferevent *bufev);
```

These functions set or return the file descriptor for a fd-based event. Only socket-based bufferevents support setfd(). Both return -1 on failure; setfd() returns 0 on success.

The bufferevent_setfd() function was introduced in Libevent 1.4.4; the bufferevent_getfd() function was introduced in Libevent 2.0.2-alpha.

Interface

```
struct event_base *bufferevent_get_base(struct bufferevent *bev);
```

This function returns the event_base of a bufferevent. It was introduced in 2.0.9-rc.

Interface

```
struct bufferevent *bufferevent_get_underlying(struct bufferevent *bufev);
```

This function returns the bufferevent that another bufferevent is using as a transport, if any. For information on when this situation would occur, see notes on filtering bufferevents.

This function was introduced in Libevent 2.0.2-alpha.

Manually locking and unlocking a bufferevent

As with evbuffers, sometimes you want to ensure that a number of operations on a bufferevent are all performed atomically. Libevent exposes functions that you can use to manually lock and unlock a bufferevent.

Interface

```
void bufferevent_lock(struct bufferevent *bufev);
void bufferevent_unlock(struct bufferevent *bufev);
```

Note that locking a bufferevent has no effect if the bufferevent was not given the BEV_OPT_THREADSAFE thread on creation, or if Libevent's threading support wasn't activated.

Locking the bufferevent with this function will lock its associated evbuffers as well. These functions are recursive: it is safe to lock a bufferevent for which you already hold the lock. You must, of course, call unlock once for every time that you locked the bufferevent.

These functions were introduced in Libevent 2.0.6-rc.

Obsolete bufferevent functionality

The bufferevent backend code underwent substantial revision between Libevent 1.4 and Libevent 2.0. In the old interface, it was sometimes normal to build with access to the internals of the struct bufferevent, and to use macros that relied on this access.

To make matters confusing, the old code sometimes used names for bufferevent functionality that were prefixed with "evbuffer".

Here's a brief guideline of what things used to be called before Libevent 2.0:

Current name	Old name
bufferevent_data_cb	evbuffercb
bufferevent_event_cb	everrorcb
BEV_EVENT_READING	EVBUFFER_READ
BEV_EVENT_WRITE	EVBUFFER_WRITE
BEV_EVENT_EOF	EVBUFFER_EOF
BEV_EVENT_ERROR	EVBUFFER_ERROR
BEV_EVENT_TIMEOUT	EVBUFFER_TIMEOUT
bufferevent_get_input(b)	EVBUFFER_INPUT(b)
bufferevent_get_output(b)	EVBUFFER_OUTPUT(b)

The old functions were defined in event.h, not in event2/bufferevent.h.

If you still need access to the internals of the common parts of the bufferevent struct, you can include event2/bufferevent_struct.h. We recommend against it: the contents of struct bufferevent WILL change between versions of Libevent. The macros and names in this section are available if you include event2/bufferevent_compat.h.

The interface to set up a bufferevent differed in older versions:

Interface

```
struct bufferevent *bufferevent_new(evutil_socket_t fd,
    evbuffercb readcb, evbuffercb writecb, everrorcb errorcb, void *cbarg);
int bufferevent_base_set(struct event_base *base, struct bufferevent *bufev);
```

The bufferevent_new() function creates a socket bufferevent only, and does so on the deprecated "default" event_base. Calling bufferevent_base_set adjusts the event_base of a socket bufferevent only.

Instead of setting timeouts as struct timeval, they were set as numbers of seconds:

Interface

```
void bufferevent_settimeout(struct bufferevent *bufev,
    int timeout_read, int timeout_write);
```

Finally, note that the underlying evbuffer implementation for Libevent versions before 2.0 was pretty inefficient, to the point where using bufferevents for high-performance apps was kind of questionable.

Bufferevents: advanced topics

This chapter describes some advanced features of Libevent's bufferevent implementation that aren't necessary for typical uses. If you're just learning how to use bufferevents, you should skip this chapter for now and go on to read the evbuffer chapter.

Paired bufferevents

Sometimes you have a networking program that needs to talk to itself. For example, you could have a program written to tunnel user connections over some protocol that sometimes also wants to tunnel connections *of its own* over that protocol. You could

achieve this by opening a connection to your own listening port and having your program use itself, of course, but that would waste resources by having your program talk to itself via the network stack.

Instead, you can create a pair of *paired* bufferevents such that all bytes written on one are received on the other (and vice versa), but no actual platform sockets are used.

Interface

```
int bufferevent_pair_new(struct event_base *base, int options,
    struct bufferevent *pair[2]);
```

Calling bufferevent_pair_new() sets pair[0] and pair[1] to a pair of bufferevents, each connected to the other. All the usual options are supported, except for BEV_OPT_CLOSE_ON_FREE, which has no effect, and BEV_OPT_DEFER_CALLBACKS, which is always on.

Why do bufferevent pairs need to run with callbacks deferred? It's pretty common for an operation on one element of the pair to invoke a callback that alters the bufferevent, thus invoking the other bufferevent's callbacks, and so on through many steps. When the callbacks were not deferred, this chain of calls would pretty frequently overflow the stack, starve other connections, and require all the callbacks to be reentrant.

Paired bufferevents support flushing; setting the mode argument to either either BEV_NORMAL or BEV_FLUSH forces all the relevant data to get transferred from one bufferevent in the pair to the other, ignoring the watermarks that would otherwise restrict it. Setting mode to BEV_FINISHED additionally generates an EOF event on the opposite bufferevent.

Freeing either member of the pair *does not* automatically free the other or generate an EOF event; it just makes the other member of the pair become unlinked. Once the bufferevent is unlinked, it will no longer successfully read or write data or generate any events.

Interface

```
struct bufferevent *bufferevent_pair_get_partner(struct bufferevent *bev)
```

Sometimes you may need to get the other member of a bufferevent pair given only one member. To do this, you can invoke the bufferevent_pair_get_partner() function. It will return the other member of the pair if *bev* is a member of a pair, and the other member still exists. Otherwise, it returns NULL.

Bufferevent pairs were new in Libevent 2.0.1-alpha; the bufferevent_pair_get_partner() function was introduced in Libevent 2.0.6.

Filtering bufferevents

Sometimes you want to transform all the data passing through a bufferevent object. You could do this to add a compression layer, or wrap a protocol in another protocol for transport.

Interface

```
enum bufferevent_filter_result {
        BEV_OK = 0,
        BEV_NEED_MORE = 1,
        BEV_ERROR = 2
};
typedef enum bufferevent_filter_result (*bufferevent_filter_cb)(
    struct evbuffer *source, struct evbuffer *destination, ev_ssize_t dst_limit,
    enum bufferevent_flush_mode mode, void *ctx);


struct bufferevent *bufferevent_filter_new(struct bufferevent *underlying,
        bufferevent_filter_cb input_filter,
        bufferevent_filter_cb output_filter,
        int options,
        void (*free_context)(void *),
        void *ctx);
```

The bufferevent_filter_new() function creates a new filtering bufferevent, wrapped around an existing "underlying" bufferevent. All data received via the underlying bufferevent is transformed with the "input" filter before arriving at the filtering bufferevent, and all data sent via the filtering bufferevent is transformed with an "output" filter before being sent out to the underlying bufferevent.

Adding a filter to an underlying bufferevent replaces the callbacks on the underlying bufferevent. You can still add callbacks to the underlying bufferevent's evbuffers, but you can't set the callbacks on the bufferevent itself if you want the filter to still work.

The *input_filter* and *output_filter* functions are described below. All the usual options are supported in *options*. If BEV_OPT_CLOSE_O is set, then freeing the filtering bufferevent also frees the underlying bufferevent. The *ctx* field is an arbitrary pointer passed to the filter functions; if a *free_context* function is provided, it is called on *ctx* just before the filtering bufferevent is closed.

The input filter function will be called whenever there is new readable data on the underlying input buffer. The output filter function is called whenever there is new writable data on the filter's output buffer. Each one receives a pair of evbuffers: a *source* evbuffer to read data from, and a *destination* evbuffer to write data to. The *dst_limit* argument describes the upper bound of bytes to add to *destination*. The filter function is allowed to ignore this value, but doing so might violate high-water marks or rate limits. If *dst_limit* is -1, there is no limit. The *mode* parameter tells the filter how aggressive to be in writing. If it is BEV_NORMAL, then it should write as much as can be conveniently transformed. The BEV_FLUSH value means to write as much as possible, and BEV_FINISHED means that the filtering function should additionally do any cleanup necessary at the end of the stream. Finally, the filter function's *ctx* argument is a void pointer as provided to the bufferevent_filter_new() constructor.

Filter functions must return BEV_OK if any data was successfully written to the destination buffer, BEV_NEED_MORE if no more data can be written to the destination buffer without getting more input or using a different flush mode, and BEV_ERROR if there is a non-recoverable error on the filter.

Creating the filter enables both reading and writing on the underlying bufferevent. You do not need to manage reads/writes on your own: the filter will suspend reading on the underlying bufferevent for you whenever it doesn't want to read. For 2.0.8-rc and later, it is permissible to enable/disable reading and writing on the underlying bufferevent independently from the filter. If you do this, though, you may keep the filter from successfully getting the data it wants.

You don't need to specify both an input filter and an output filter: any filter you omit is replaced with one that passes data on without transforming it.

Limiting maximum single read/write size

By default, bufferevents won't read or write the maximum possible amount of bytes on each invocation of the event loop; doing so can lead to weird unfair behaviors and resource starvation. On the other hand, the defaults might not be reasonable for all situations.

Interface

```
int bufferevent_set_max_single_read(struct bufferevent *bev, size_t size);
int bufferevent_set_max_single_write(struct bufferevent *bev, size_t size);

ev_ssize_t bufferevent_get_max_single_read(struct bufferevent *bev);
ev_ssize_t bufferevent_get_max_single_write(struct bufferevent *bev);
```

The two "set" functions replace the current read and write maxima respectively. If the *size* value is 0 or above EV_SSIZE_MAX, they instead set the maxima to the default value. These functions return 0 on success and -1 on failure.

The two "get" functions return the current per-loop read and write maxima respectively.

These functions were added in 2.1.1-alpha.

Bufferevents and Rate-limiting

Some programs want to limit the amount of bandwidth used for any single bufferevent, or for a group of bufferevents. Libevent 2.0.4-alpha and Libevent 2.0.5-alpha added a basic facility to put caps on individual bufferevents, or to assign bufferevents to a rate-limited group.

The rate-limiting model

Libevent's rate-limiting uses a *token bucket* algorithm to decide how many bytes to read or write at a time. Every rate-limited object, at any given time, has a "read bucket" and a "write bucket", the sizes of which determine how many bytes the object is allowed to read or write immediately. Each bucket has a refill rate, a maximum burst size, and a timing unit or "tick". Whenever the timing unit elapses, the bucket is refilled proportionally to the refill rate—but if would become fuller than its burst size, any excess bytes are lost.

Thus, the refill rate determines the maximum average rate at which the object will send or receive bytes, and the burst size determines the largest number of bytes that will be sent or received in a single burst. The timing unit determines the smoothness of the traffic.

Setting a rate limit on a bufferevent

Interface

```
#define EV_RATE_LIMIT_MAX EV_SSIZE_MAX
struct ev_token_bucket_cfg;
struct ev_token_bucket_cfg *ev_token_bucket_cfg_new(
        size_t read_rate, size_t read_burst,
        size_t write_rate, size_t write_burst,
        const struct timeval *tick_len);
void ev_token_bucket_cfg_free(struct ev_token_bucket_cfg *cfg);
int bufferevent_set_rate_limit(struct bufferevent *bev,
    struct ev_token_bucket_cfg *cfg);
```

An *ev_token_bucket_cfg* structure represents the configuration values for a pair of token buckets used to limit reading and writing on a single bufferevent or group of bufferevents. To create one, call the ev_token_bucket_cfg_new function and provide the maximum average read rate, the maximum read burst, the maximum write rate, the maximum write burst, and the length of a tick. If the *tick_len* argument is NULL, the length of a tick defaults to one second. The function may return NULL on error.

Note that the *read_rate* and *write_rate* arguments are scaled in units of bytes per tick. That is, if the tick is one tenth of a second, and *read_rate* is 300, then the maximum average read rate is 3000 bytes per second. Rate and burst values over EV_RATE_LIMIT_MAX are not supported.

To limit a bufferevent's transfer rate, call bufferevent_set_rate_limit() on it with an ev_token_bucket_cfg. The function returns 0 on success, and -1 on failure. You can give any number of bufferevents the same ev_token_bucket_cfg. To remove a bufferevent's rate limits, call bufferevent_set_rate_limit(), passing NULL for the *cfg* parameter.

To free an ev_token_bucket_cfg, call ev_token_bucket_cfg_free(). Note that it is NOT currently safe to do this until no bufferevents are using the ev_token_bucket_cfg.

Setting a rate limit on a group of bufferevents

You can assign bufferevents to a *rate limiting group* if you want to limit their total bandwidth usage.

Interface

```
struct bufferevent_rate_limit_group;

struct bufferevent_rate_limit_group *bufferevent_rate_limit_group_new(
        struct event_base *base,
        const struct ev_token_bucket_cfg *cfg);
int bufferevent_rate_limit_group_set_cfg(
        struct bufferevent_rate_limit_group *group,
        const struct ev_token_bucket_cfg *cfg);
void bufferevent_rate_limit_group_free(struct bufferevent_rate_limit_group *);
int bufferevent_add_to_rate_limit_group(struct bufferevent *bev,
    struct bufferevent_rate_limit_group *g);
int bufferevent_remove_from_rate_limit_group(struct bufferevent *bev);
```

To construct a rate limiting group, call bufferevent_rate_limit_group() with an event_base and an initial ev_token_bucket_cfg. You can add bufferevents to the group with bufferevent_add_to_rate_limit_group() and bufferevent_remove_from_rate_limit_group(); these functions return 0 on success and -1 on error.

A single bufferevent can be a member of no more than one rate limiting group at a time. A bufferevent can have both an individual rate limit (as set with bufferevent_set_rate_limit()) and a group rate limit. When both limits are set, the lower limit for each bufferevent applies.

You can change the rate limit for an existing group by calling bufferevent_rate_limit_group_set_cfg(). It returns 0 on success and -1 on failure. The bufferevent_rate_limit_group_free() function frees a rate limit group and removes all of its members.

As of version 2.0, Libevent's group rate limiting tries to be fair on aggregate, but the implementation can be unfair on very small timescales. If you care strongly about scheduling fairness, please help out with patches for future versions.

Inspecting current rate-limit values

Sometimes your code may want to inspect the current rate limits that apply for a given bufferevent or group. Libevent provides some functions to do so.

Interface

```
ev_ssize_t bufferevent_get_read_limit(struct bufferevent *bev);
ev_ssize_t bufferevent_get_write_limit(struct bufferevent *bev);
ev_ssize_t bufferevent_rate_limit_group_get_read_limit(
        struct bufferevent_rate_limit_group *);
ev_ssize_t bufferevent_rate_limit_group_get_write_limit(
        struct bufferevent_rate_limit_group *);
```

The above functions return the current size, in bytes, of a bufferevent's or a group's read or write token buckets. Note that these values can be negative if a bufferevent has been forced to exceed its allocations. (Flushing the bufferevent can do this.)

Interface

```
ev_ssize_t bufferevent_get_max_to_read(struct bufferevent *bev);
ev_ssize_t bufferevent_get_max_to_write(struct bufferevent *bev);
```

These functions return the number of bytes that a bufferevent would be willing to read or write right now, taking into account any rate limits that apply to the bufferevent, its rate limiting group (if any), and any maximum-to-read/write-at-a-time values imposed by Libevent as a whole.

Interface

```
void bufferevent_rate_limit_group_get_totals(
    struct bufferevent_rate_limit_group *grp,
    ev_uint64_t *total_read_out, ev_uint64_t *total_written_out);
void bufferevent_rate_limit_group_reset_totals(
    struct bufferevent_rate_limit_group *grp);
```

Each bufferevent_rate_limit_group tracks the total number of bytes sent over it, in total. You can use this to track total usage by a number of bufferevents in the group. Calling bufferevent_rate_limit_group_get_totals() on a group sets **total_read_out* and **total_written_out* to the total number of bytes read and written on a bufferevent group respectively. These totals start at 0 when the group is created, and reset to 0 whenever bufferevent_rate_limit_group_reset_totals() is called on a group.

Manually adjusting rate limits

For programs with really complex needs, you might want to adjust the current values of a token bucket. You might want to do this, for example, if your program is generating traffic in some way that isn't via a bufferevent.

Interface

```
int bufferevent_decrement_read_limit(struct bufferevent *bev, ev_ssize_t decr);
int bufferevent_decrement_write_limit(struct bufferevent *bev, ev_ssize_t decr);
int bufferevent_rate_limit_group_decrement_read(
        struct bufferevent_rate_limit_group *grp, ev_ssize_t decr);
int bufferevent_rate_limit_group_decrement_write(
        struct bufferevent_rate_limit_group *grp, ev_ssize_t decr);
```

These functions decrement a current read or write bucket in a bufferevent or rate limiting group. Note that the decrements are signed: if you want to increment a bucket, pass a negative value.

Setting the smallest share possible in a rate-limited group

Frequently, you don't want to divide the bytes available in a rate-limiting group up evenly among all bufferevents in every tick. For example, if you had 10,000 active bufferevents in a rate-limiting group with 10,000 bytes available for writing every tick, it wouldn't be efficient to let each bufferevent write only 1 byte per tick, due to the overheads of system calls and TCP headers.

To solve this, each rate-limiting group has a notion of its "minimum share". In the situation above, instead of every bufferevent being allowed to write 1 byte per tick, 10,000/SHARE bufferevents will be allowed to write SHARE bytes each every tick, and the rest will be allowed to write nothing. Which bufferevents are allowed to write first is chosen randomly each tick.

The default minimum share is chosen to give decent performance, and is currently (as of 2.0.6-rc) set to 64. You can adjust this value with the following function:

Interface

```
int bufferevent_rate_limit_group_set_min_share(
        struct bufferevent_rate_limit_group *group, size_t min_share);
```

Setting the min_share to 0 disables the minimum-share code entirely.

Libevent's rate-limiting has had minimum shares since it was first introduced. The function to change them was first exposed in Libevent 2.0.6-rc.

Limitations of the rate-limiting implementation

As of Libevent 2.0, there are some limitations to the rate-limiting implementation that you should know.

- Not every bufferevent type supports rate limiting well, or at all.
- Bufferevent rate limiting groups cannot nest, and a bufferevent can only be in a single rate limiting group at a time.
- The rate limiting implementation only counts bytes transferred in TCP packets as data, doesn't include TCP headers.
- The read-limiting implementation relies on the TCP stack noticing that the application is only consuming data at a certain rate, and pushing back on the other side of the TCP connection when its buffers get full.
- Some implementations of bufferevents (particularly the windows IOCP implementation) can over-commit.
- Buckets start out with one full tick's worth of traffic. This means that a bufferevent can start reading or writing immediately, and not wait until a full tick has passed. It also means, though, that a bufferevent that has been rate limited for N.1 ticks can potentially transfer N+1 ticks worth of traffic.
- Ticks cannot be smaller than 1 millisecond, and all fractions of a millisecond are ignored.

/// TODO: Write an example for rate-limiting

Bufferevents and SSL

Bufferevents can use the OpenSSL library to implement the SSL/TLS secure transport layer. Because many applications don't need or want to link OpenSSL, this functionality is implemented in a separate library installed as "libevent_openssl". Future versions of Libevent could add support for other SSL/TLS libraries such as NSS or GnuTLS, but right now OpenSSL is all that's there.

OpenSSL functionality was introduced in Libevent 2.0.3-alpha, though it didn't work so well before Libevent 2.0.5-beta or Libevent 2.0.6-rc.

This section is not a tutorial on OpenSSL, SSL/TLS, or cryptography in general.

These functions are all declared in the header "event2/bufferevent_ssl.h".

Setting up and using an OpenSSL-based bufferevent

Interface

```
enum bufferevent_ssl_state {
        BUFFEREVENT_SSL_OPEN = 0,
        BUFFEREVENT_SSL_CONNECTING = 1,
        BUFFEREVENT_SSL_ACCEPTING = 2
};

struct bufferevent *
bufferevent_openssl_filter_new(struct event_base *base,
    struct bufferevent *underlying,
    SSL *ssl,
    enum bufferevent_ssl_state state,
    int options);

struct bufferevent *
bufferevent_openssl_socket_new(struct event_base *base,
    evutil_socket_t fd,
    SSL *ssl,
    enum bufferevent_ssl_state state,
    int options);
```

You can create two kinds of SSL bufferevents: a filter-based bufferevent that communicates over another underlying bufferevent, or a socket-based bufferevent that tells OpenSSL to communicate with the network directly over. In either case, you must provide an SSL object and a description of the SSL object's state. The state should be BUFFEREVENT_SSL_CONNECTING if the SSL is currently performing negotiation as a client, BUFFEREVENT_SSL_ACCEPTING if the SSL is currently performing negotiation as a server, or BUFFEREVENT_SSL_OPEN if the SSL handshake is done.

The usual options are accepted; BEV_OPT_CLOSE_ON_FREE makes the SSL object and the underlying fd or bufferevent get closed when the openssl bufferevent itself is closed.

Once the handshake is complete, the new bufferevent's event callback gets invoked with BEV_EVENT_CONNECTED in flags.

If you're creating a socket-based bufferevent and the SSL object already has a socket set, you do not need to provide the socket yourself: just pass -1. You can also set the fd later with bufferevent_setfd().

/// TODO: Remove this once bufferevent_shutdown() API has been finished.

Note that when BEV_OPT_CLOSE_ON_FREE is set on a SSL bufferevent, a clean shutdown will not be performed on the SSL connection. This has two problems: first, the connection will seem to have been "broken" by the other side, rather than having been closed cleanly: the other party will not be able to tell whether you closed the connection, or whether it was broken by an attacker or third party. Second, OpenSSL will treat the session as "bad", and removed from the session cache. This can cause significant performance degradation on SSL applications under load.

Currently the only workaround is to do lazy SSL shutdowns manually. While this breaks the TLS RFC, it will make sure that sessions will stay in cache once closed. The following code implements this workaround.

Example

```
SSL *ctx = bufferevent_openssl_get_ssl(bev);

/*
 * SSL_RECEIVED_SHUTDOWN tells SSL_shutdown to act as if we had already
 * received a close notify from the other end.  SSL_shutdown will then
 * send the final close notify in reply.  The other end will receive the
 * close notify and send theirs.  By this time, we will have already
 * closed the socket and the other end's real close notify will never be
 * received.  In effect, both sides will think that they have completed a
 * clean shutdown and keep their sessions valid.  This strategy will fail
 * if the socket is not ready for writing, in which case this hack will
 * lead to an unclean shutdown and lost session on the other end.
 */
SSL_set_shutdown(ctx, SSL_RECEIVED_SHUTDOWN);
SSL_shutdown(ctx);
bufferevent_free(bev);
```

Interface

```
SSL *bufferevent_openssl_get_ssl(struct bufferevent *bev);
```

This function returns the SSL object used by an OpenSSL bufferevent, or NULL if *bev* is not an OpenSSL-based bufferevent.

Interface

```
unsigned long bufferevent_get_openssl_error(struct bufferevent *bev);
```

This function returns the first pending OpenSSL error for a given bufferevent's operations, or 0 if there was no pending error. The error format is as returned by ERR_get_error() in the openssl library.

Interface

```
int bufferevent_ssl_renegotiate(struct bufferevent *bev);
```

Calling this function tells the SSL to renegotiate, and the bufferevent to invoke appropriate callbacks. This is an advanced topic; you should generally avoid it unless you really know what you're doing, especially since many SSL versions have had known security issues related to renegotiation.

Interface

```
int bufferevent_openssl_get_allow_dirty_shutdown(struct bufferevent *bev);
void bufferevent_openssl_set_allow_dirty_shutdown(struct bufferevent *bev,
    int allow_dirty_shutdown);
```

All good versions of the SSL protocol (that is, SSLv3 and all TLS versions) support an authenticated shutdown operation that enables the parties to distinguish an intentional close from an accidental or maliciously induced termination in the underling buffer. By default, we treat anything besides a proper shutdown as an error on the connection. If the allow_dirty_shutdown flag is set to 1, however, we treat a close in the connection as a BEV_EVENT_EOF.

The allow_dirty_shutdown functions were added in Libevent 2.1.1-alpha.

Example: A simple SSL-based echo server

```
/* Simple echo server using OpenSSL bufferevents */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

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

```
#include <openssl/err.h>
#include <openssl/rand.h>

#include <event.h>
#include <event2/listener.h>
#include <event2/bufferevent_ssl.h>

static void
ssl_readcb(struct bufferevent * bev, void * arg)
{
    struct evbuffer *in = bufferevent_get_input(bev);

    printf("Received %zu bytes\n", evbuffer_get_length(in));
    printf("----- data ----\n");
    printf("%.*s\n", (int)evbuffer_get_length(in), evbuffer_pullup(in, -1));

    bufferevent_write_buffer(bev, in);
}

static void
ssl_acceptcb(struct evconnlistener *serv, int sock, struct sockaddr *sa,
             int sa_len, void *arg)
{
    struct event_base *evbase;
    struct bufferevent *bev;
    SSL_CTX *server_ctx;
    SSL *client_ctx;

    server_ctx = (SSL_CTX *)arg;
    client_ctx = SSL_new(server_ctx);
    evbase = evconnlistener_get_base(serv);

    bev = bufferevent_openssl_socket_new(evbase, sock, client_ctx,
                                         BUFFEREVENT_SSL_ACCEPTING,
                                         BEV_OPT_CLOSE_ON_FREE);

    bufferevent_enable(bev, EV_READ);
    bufferevent_setcb(bev, ssl_readcb, NULL, NULL, NULL);
}

static SSL_CTX *
evssl_init(void)
{
    SSL_CTX  *server_ctx;

    /* Initialize the OpenSSL library */
    SSL_load_error_strings();
    SSL_library_init();
    /* We MUST have entropy, or else there's no point to crypto. */
    if (!RAND_poll())
        return NULL;

    server_ctx = SSL_CTX_new(SSLv23_server_method());

    if (! SSL_CTX_use_certificate_chain_file(server_ctx, "cert") ||
        ! SSL_CTX_use_PrivateKey_file(server_ctx, "pkey", SSL_FILETYPE_PEM)) {
        puts("Couldn't read 'pkey' or 'cert' file.  To generate a key\n"
           "and self-signed certificate, run:\n"
           "  openssl genrsa -out pkey 2048\n"
           "  openssl req -new -key pkey -out cert.req\n"
           "  openssl x509 -req -days 365 -in cert.req -signkey pkey -out cert");
        return NULL;
```

```
    }
    SSL_CTX_set_options(server_ctx, SSL_OP_NO_SSLv2);

    return server_ctx;
}

int
main(int argc, char **argv)
{
    SSL_CTX *ctx;
    struct evconnlistener *listener;
    struct event_base *evbase;
    struct sockaddr_in sin;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_port = htons(9999);
    sin.sin_addr.s_addr = htonl(0x7f000001); /* 127.0.0.1 */

    ctx = evssl_init();
    if (ctx == NULL)
        return 1;
    evbase = event_base_new();
    listener = evconnlistener_new_bind(
                        evbase, ssl_acceptcb, (void *)ctx,
                        LEV_OPT_CLOSE_ON_FREE | LEV_OPT_REUSEABLE, 1024,
                        (struct sockaddr *)&sin, sizeof(sin));

    event_base_loop(evbase, 0);

    evconnlistener_free(listener);
    SSL_CTX_free(ctx);

    return 0;
}
```

Some notes on threading and OpenSSL

The built in threading mechanisms of Libevent do not cover OpenSSL locking. Since OpenSSL uses a myriad of global variables, you must still configure OpenSSL to be thread safe. While this process is outside the scope of Libevent, this topic comes up enough to warrant discussion.

Example: A very simple example of how to enable thread safe OpenSSL

```
/*
 * Please refer to OpenSSL documentation to verify you are doing this correctly,
 * Libevent does not guarantee this code is the complete picture, but to be used
 * only as an example.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <pthread.h>
#include <openssl/ssl.h>
#include <openssl/crypto.h>

pthread_mutex_t * ssl_locks;
int ssl_num_locks;

/* Implements a thread-ID function as requied by openssl */
```

```
static unsigned long
get_thread_id_cb(void)
{
    return (unsigned long)pthread_self();
}

static void
thread_lock_cb(int mode, int which, const char * f, int l)
{
    if (which < ssl_num_locks) {
        if (mode & CRYPTO_LOCK) {
            pthread_mutex_lock(&(ssl_locks[which]));
        } else {
            pthread_mutex_unlock(&(ssl_locks[which]));
        }
    }
}

int
init_ssl_locking(void)
{
    int i;

    ssl_num_locks = CRYPTO_num_locks();
    ssl_locks = malloc(ssl_num_locks * sizeof(pthread_mutex_t));
    if (ssl_locks == NULL)
        return -1;

    for (i = 0; i < ssl_num_locks; i++) {
        pthread_mutex_init(&(ssl_locks[i]), NULL);
    }

    CRYPTO_set_id_callback(get_thread_id_cb);
    CRYPTO_set_locking_callback(thread_lock_cb);

    return 0;
}
```


Evbuffers: utility functionality for buffered IO

Libevent's evbuffer functionality implements a queue of bytes, optimized for adding data to the end and removing it from the front.

Evbuffers are meant to be generally useful for doing the "buffer" part of buffered network IO. They do not provide functions to schedule the IO or trigger the IO when it's ready: that is what bufferevents do.

The functions in this chapter are declared in event2/buffer.h unless otherwise noted.

Creating or freeing an evbuffer

Interface

```
struct evbuffer *evbuffer_new(void);
void evbuffer_free(struct evbuffer *buf);
```

These functions should be relatively clear: evbuffer_new() allocates and returns a new empty evbuffer, and evbuffer_free() deletes one and all of its contents.

These functions have existed since Libevent 0.8.

Evbuffers and Thread-safety

Interface

```
int evbuffer_enable_locking(struct evbuffer *buf, void *lock);
void evbuffer_lock(struct evbuffer *buf);
void evbuffer_unlock(struct evbuffer *buf);
```

By default, it is not safe to access an evbuffer from multiple threads at once. If you need to do this, you can call evbuffer_enable_locking() on the evbuffer. If its *lock* argument is NULL, Libevent allocates a new lock using the lock creation function that was provided to evthread_set_lock_creation_callback. Otherwise, it uses the argument as the lock.

The evbuffer_lock() and evbuffer_unlock() functions acquire and release the lock on an evbuffer respectively. You can use them to make a set of operations atomic. If locking has not been enabled on the evbuffer, these functions do nothing.

(Note that you do not need to call evbuffer_lock() and evbuffer_unlock() around *individual* operations: if locking is enabled on the evbuffer, individual operations are already atomic. You only need to lock the evbuffer manually when you have more than one operation that need to execute without another thread butting in.)

These functions were all introduced in Libevent 2.0.1-alpha.

Inspecting an evbuffer

Interface

```
size_t evbuffer_get_length(const struct evbuffer *buf);
```

This function returns the number of bytes stored in an evbuffer.

It was introduced in Libevent 2.0.1-alpha.

Interface

```
size_t evbuffer_get_contiguous_space(const struct evbuffer *buf);
```

This function returns the number of bytes stored contiguously at the front of the evbuffer. The bytes in an evbuffer may be stored in multiple separate chunks of memory; this function returns the number of bytes currently stored in the *first* chunk.

It was introduced in Libevent 2.0.1-alpha.

Adding data to an evbuffer: basics

Interface

```
int evbuffer_add(struct evbuffer *buf, const void *data, size_t datlen);
```

This function appends the *datlen* bytes in *data* to the end of *buf*. It returns 0 on success, and -1 on failure.

Interface

```
int evbuffer_add_printf(struct evbuffer *buf, const char *fmt, ...)
int evbuffer_add_vprintf(struct evbuffer *buf, const char *fmt, va_list ap);
```

These functions append formatted data to the end of *buf*. The format argument and other remaining arguments are handled as if by the C library functions "printf" and "vprintf" respectively. The functions return the number of bytes appended.

Interface

```
int evbuffer_expand(struct evbuffer *buf, size_t datlen);
```

This function alters the last chunk of memory in the buffer, or adds a new chunk, such that the buffer is now large enough to contain datlen bytes without any further allocations.

Examples

```
/* Here are two ways to add "Hello world 2.0.1" to a buffer. */
/* Directly: */
evbuffer_add(buf, "Hello world 2.0.1", 17);

/* Via printf: */
evbuffer_add_printf(buf, "Hello %s %d.%d.%d", "world", 2, 0, 1);
```

The evbuffer_add() and evbuffer_add_printf() functions were introduced in Libevent 0.8; evbuffer_expand() was in Libevent 0.9, and evbuffer_add_vprintf() first appeared in Libevent 1.1.

Moving data from one evbuffer to another

For efficiency, Libevent has optimized functions for moving data from one evbuffer to another.

Interface

```
int evbuffer_add_buffer(struct evbuffer *dst, struct evbuffer *src);
int evbuffer_remove_buffer(struct evbuffer *src, struct evbuffer *dst,
    size_t datlen);
```

The evbuffer_add_buffer() function moves all data from *src* to the end of *dst*. It returns 0 on success, -1 on failure.

The evbuffer_remove_buffer() function moves exactly *datlen* bytes from *src* to the end of *dst*, copying as little as possible. If there are fewer than *datlen* bytes to move, it moves all the bytes. It returns the number of bytes moved.

We introduced evbuffer_add_buffer() in Libevent 0.8; evbuffer_remove_buffer() was new in Libevent 2.0.1-alpha.

Adding data to the front of an evbuffer

Interface

```
int evbuffer_prepend(struct evbuffer *buf, const void *data, size_t size);
int evbuffer_prepend_buffer(struct evbuffer *dst, struct evbuffer* src);
```

These functions behave as evbuffer_add() and evbuffer_add_buffer() respectively, except that they move data to the *front* of the destination buffer.

These functions should be used with caution, and never on an evbuffer shared with a bufferevent. They were new in Libevent 2.0.1-alpha.

Rearranging the internal layout of an evbuffer

Sometimes you want to peek at the first N bytes of data in the front of an evbuffer, and see it as a contiguous array of bytes. To do this, you must first ensure that the front of the buffer really *is* contiguous.

Interface

```
unsigned char *evbuffer_pullup(struct evbuffer *buf, ev_ssize_t size);
```

The evbuffer_pullup() function "linearizes" the first *size* bytes of *buf*, copying or moving them as needed to ensure that they are all contiguous and occupying the same chunk of memory. If *size* is negative, the function linearizes the entire buffer. If *size* is greater than the number of bytes in the buffer, the function returns NULL. Otherwise, evbuffer_pullup() returns a pointer to the first byte in buf.

Calling evbuffer_pullup() with a large size can be quite slow, since it potentially needs to copy the entire buffer's contents.

Example

```
#include <event2/buffer.h>
#include <event2/util.h>

#include <string.h>

int parse_socks4(struct evbuffer *buf, ev_uint16_t *port, ev_uint32_t *addr)
{
    /* Let's parse the start of a SOCKS4 request!  The format is easy:
     * 1 byte of version, 1 byte of command, 2 bytes destport, 4 bytes of
     * destip. */
    unsigned char *mem;

    mem = evbuffer_pullup(buf, 8);

    if (mem == NULL) {
        /* Not enough data in the buffer */
        return 0;
    } else if (mem[0] != 4 || mem[1] != 1) {
        /* Unrecognized protocol or command */
        return -1;
    } else {
        memcpy(port, mem+2, 2);
        memcpy(addr, mem+4, 4);
        *port = ntohs(*port);
        *addr = ntohl(*addr);
        /* Actually remove the data from the buffer now that we know we
           like it. */
        evbuffer_drain(buf, 8);
        return 1;
    }
}
```

Note Calling evbuffer_pullup() with size equal to the value returned by evbuffer_get_contiguous_space() will not result in any data being copied or moved.

The evbuffer_pullup() function was new in Libevent 2.0.1-alpha: previous versions of Libevent always kept evbuffer data contiguous, regardless of the cost.

Removing data from an evbuffer

Interface

```
int evbuffer_drain(struct evbuffer *buf, size_t len);
int evbuffer_remove(struct evbuffer *buf, void *data, size_t datlen);
```

The evbuffer_remove() function copies and removes the first *datlen* bytes from the front of *buf* into the memory at *data*. If there are fewer than *datlen* bytes available, the function copies all the bytes there are. The return value is -1 on failure, and is otherwise the number of bytes copied.

The evbuffer_drain() function behaves as evbuffer_remove(), except that it does not copy the data: it just removes it from the front of the buffer. It returns 0 on success and -1 on failure.

Libevent 0.8 introduced evbuffer_drain(); evbuffer_remove() appeared in Libevent 0.9.

Copying data out from an evbuffer

Sometimes you want to get a copy of the data at the start of a buffer without draining it. For example, you might want to see whether a complete record of some kind has arrived, without draining any of the data (as evbuffer_remove would do), or rearranging the buffer internally (as evbuffer_pullup() would do.)

Interface

```
ev_ssize_t evbuffer_copyout(struct evbuffer *buf, void *data, size_t datlen);
ev_ssize_t evbuffer_copyout_from(struct evbuffer *buf,
     const struct evbuffer_ptr *pos,
     void *data_out, size_t datlen);
```

The evbuffer_copyout() behaves just like evbuffer_remove(), but does not drain any data from the buffer. That is, it copies the first *datlen* bytes from the front of *buf* into the memory at *data*. If there are fewer than *datlen* bytes available, the function copies all the bytes there are. The return value is -1 on failure, and is otherwise the number of bytes copied.

The evbuffer_copyout_from() function behaves like evbuffer_copyout(), but instead of copying bytes from the front of the buffer, it copies them beginning at the position provided in *pos*. See "Searching within an evbuffer" below for information on the evbuffer_ptr structure.

If copying data from the buffer is too slow, use evbuffer_peek() instead.

Example

```
#include <event2/buffer.h>
#include <event2/util.h>
#include <stdlib.h>
#include <stdlib.h>

int get_record(struct evbuffer *buf, size_t *size_out, char **record_out)
{
    /* Let's assume that we're speaking some protocol where records
       contain a 4-byte size field in network order, followed by that
       number of bytes.  We will return 1 and set the 'out' fields if we
       have a whole record, return 0 if the record isn't here yet, and
       -1 on error.  */
    size_t buffer_len = evbuffer_get_length(buf);
    ev_uint32_t record_len;
    char *record;

    if (buffer_len < 4)
       return 0; /* The size field hasn't arrived. */

   /* We use evbuffer_copyout here so that the size field will stay on
       the buffer for now. */
    evbuffer_copyout(buf, &record_len, 4);
    /* Convert len_buf into host order. */
    record_len = ntohl(record_len);
    if (buffer_len < record_len + 4)
        return 0; /* The record hasn't arrived */

    /* Okay, _now_ we can remove the record. */
    record = malloc(record_len);
```

```
    if (record == NULL)
        return -1;

    evbuffer_drain(buf, 4);
    evbuffer_remove(buf, record, record_len);

    *record_out = record;
    *size_out = record_len;
    return 1;
}
```

The evbuffer_copyout() function first appeared in Libevent 2.0.5-alpha; evbuffer_copyout_from() was added in Libevent 2.1.1-alpha.

Line-oriented input

Interface

```
enum evbuffer_eol_style {
        EVBUFFER_EOL_ANY,
        EVBUFFER_EOL_CRLF,
        EVBUFFER_EOL_CRLF_STRICT,
        EVBUFFER_EOL_LF,
        EVBUFFER_EOL_NUL
};
char *evbuffer_readln(struct evbuffer *buffer, size_t *n_read_out,
    enum evbuffer_eol_style eol_style);
```

Many Internet protocols use line-based formats. The evbuffer_readln() function extracts a line from the front of an evbuffer and returns it in a newly allocated NUL-terminated string. If *n_read_out* is not NULL, **n_read_out* is set to the number of bytes in the string returned. If there is not a whole line to read, the function returns NULL. The line terminator is not included in the copied string.

The evbuffer_readln() function understands 4 line termination formats:

EVBUFFER_EOL_LF
: The end of a line is a single linefeed character. (This is also known as "\n". It is ASCII value is 0x0A.)

EVBUFFER_EOL_CRLF_STRICT
: The end of a line is a single carriage return, followed by a single linefeed. (This is also known as "\r\n". The ASCII values are 0x0D 0x0A).

EVBUFFER_EOL_CRLF
: The end of the line is an optional carriage return, followed by a linefeed. (In other words, it is either a "\r\n" or a "\n".) This format is useful in parsing text-based Internet protocols, since the standards generally prescribe a "\r\n" line-terminator, but nonconformant clients sometimes say just "\n".

EVBUFFER_EOL_ANY
: The end of line is any sequence of any number of carriage return and linefeed characters. This format is not very useful; it exists mainly for backward compatibility.

EVBUFFER_EOL_NUL
: The end of line is a single byte with the value 0 — that is, an ASCII NUL.

(Note that if you used event_set_mem_functions() to override the default malloc, the string returned by evbuffer_readln will be allocated by the malloc-replacement you specified.)

Example

```
char *request_line;
size_t len;

request_line = evbuffer_readln(buf, &len, EVBUFFER_EOL_CRLF);
if (!request_line) {
    /* The first line has not arrived yet. */
} else {
    if (!strncmp(request_line, "HTTP/1.0 ", 9)) {
        /* HTTP 1.0 detected ... */
    }
    free(request_line);
}
```

The evbuffer_readln() interface is available in Libevent 1.4.14-stable and later. EVBUFFER_EOL_NUL was added in Libevent 2.1.1-alpha.

Searching within an evbuffer

The evbuffer_ptr structure points to a location within an evbuffer, and contains data that you can use to iterate through an evbuffer.

Interface

```
struct evbuffer_ptr {
        ev_ssize_t pos;
        struct {
                /* internal fields */
        } _internal;
};
```

The *pos* field is the only public field; the others should not be used by user code. It indicates a position in the evbuffer as an offset from the start.

Interface

```
struct evbuffer_ptr evbuffer_search(struct evbuffer *buffer,
    const char *what, size_t len, const struct evbuffer_ptr *start);
struct evbuffer_ptr evbuffer_search_range(struct evbuffer *buffer,
    const char *what, size_t len, const struct evbuffer_ptr *start,
    const struct evbuffer_ptr *end);
struct evbuffer_ptr evbuffer_search_eol(struct evbuffer *buffer,
    struct evbuffer_ptr *start, size_t *eol_len_out,
    enum evbuffer_eol_style eol_style);
```

The evbuffer_search() function scans the buffer for an occurrence of the *len*-character string *what*. It returns an evbuffer_ptr containing the position of the string, or -1 if the string was not found. If the *start* argument is provided, it's the position at which the search should begin; otherwise, the search is from the start of the string.

The evbuffer_search_range() function behaves as evbuffer_search, except that it only considers occurrences of *what* that occur before the evbuffer_ptr *end*.

The evbuffer_search_eol() function detects line-endings as evbuffer_readln(), but instead of copying out the line, returns an evbuffer_ptr to the start of the end-of-line characters(s). If eol_len_out is non-NULL, it is set to the length of the EOL string.

Interface

```
enum evbuffer_ptr_how {
        EVBUFFER_PTR_SET,
        EVBUFFER_PTR_ADD
};
int evbuffer_ptr_set(struct evbuffer *buffer, struct evbuffer_ptr *pos,
    size_t position, enum evbuffer_ptr_how how);
```

The evbuffer_ptr_set function manipulates the position of an evbuffer_ptr *pos* within *buffer*. If *how* is EVBUFFER_PTR_SET, the pointer is moved to an absolute position *position* within the buffer. If it is EVBUFFER_PTR_ADD, the pointer moves *position* bytes forward. This function returns 0 on success and -1 on failure.

Example

```
#include <event2/buffer.h>
#include <string.h>

/* Count the total occurrences of 'str' in 'buf'. */
int count_instances(struct evbuffer *buf, const char *str)
{
    size_t len = strlen(str);
    int total = 0;
    struct evbuffer_ptr p;

    if (!len)
        /* Don't try to count the occurrences of a 0-length string. */
        return -1;

    evbuffer_ptr_set(buf, &p, 0, EVBUFFER_PTR_SET);

    while (1) {
         p = evbuffer_search(buf, str, len, &p);
         if (p.pos < 0)
             break;
         total++;
         evbuffer_ptr_set(buf, &p, 1, EVBUFFER_PTR_ADD);
    }

    return total;
}
```

WARNING Any call that modifies an evbuffer or its layout invalidates all outstanding evbuffer_ptr values, and makes them unsafe to use.

These interfaces were new in Libevent 2.0.1-alpha.

Inspecting data without copying it

Sometimes, you want to read data in an evbuffer without copying it out (as evbuffer_copyout() does), and without rearranging the evbuffer's internal memory (as evbuffer_pullup() does). Sometimes you might want to see data in the middle of an evbuffer.

You can do this with:

Interface

```
struct evbuffer_iovec {
        void *iov_base;
        size_t iov_len;
};

int evbuffer_peek(struct evbuffer *buffer, ev_ssize_t len,
    struct evbuffer_ptr *start_at,
    struct evbuffer_iovec *vec_out, int n_vec);
```

When you call evbuffer_peek(), you give it an array of evbuffer_iovec structures in *vec_out*. The array's length is *n_vec*. It sets these structures so that each one contains a pointer to a chunk of the evbuffer's internal RAM (*iov_base*), and the length of memory that is set in that chunk.

If *len* is less than 0, evbuffer_peek() tries to fill all of the evbuffer_iovec structs you have given it. Otherwise, it fills them until either they are all used, or at least *len* bytes are visible. If the function could give you all the data you asked for, it returns the

number of evbuffer_iovec structures that it actually used. Otherwise, it returns the number that it would need in order to give what you asked for.

When *ptr* is NULL, evbuffer_peek() starts at the beginning of the buffer. Otherwise, it starts at the pointer given in *ptr*.

Examples

```
{
    /* Let's look at the first two chunks of buf, and write them to stderr. */
    int n, i;
    struct evbuffer_iovec v[2];
    n = evbuffer_peek(buf, -1, NULL, v, 2);
    for (i=0; i<n; ++i) { /* There might be less than two chunks available. */
        fwrite(v[i].iov_base, 1, v[i].iov_len, stderr);
    }
}

{
    /* Let's send the first 4906 bytes to stdout via write. */
    int n, i, r;
    struct evbuffer_iovec *v;
    size_t written = 0;

    /* determine how many chunks we need. */
    n = evbuffer_peek(buf, 4096, NULL, NULL, 0);
    /* Allocate space for the chunks.  This would be a good time to use
       alloca() if you have it. */
    v = malloc(sizeof(struct evbuffer_iovec)*n);
    /* Actually fill up v. */
    n = evbuffer_peek(buf, 4096, NULL, v, n);
    for (i=0; i<n; ++i) {
        size_t len = v[i].iov_len;
        if (written + len > 4096)
            len = 4096 - written;
        r = write(1 /* stdout */, v[i].iov_base, len);
        if (r<=0)
            break;
        /* We keep track of the bytes written separately; if we don't,
           we may write more than 4096 bytes if the last chunk puts
           us over the limit. */
        written += len;
    }
    free(v);
}

{
    /* Let's get the first 16K of data after the first occurrence of the
       string "start\n", and pass it to a consume() function. */
    struct evbuffer_ptr ptr;
    struct evbuffer_iovec v[1];
    const char s[] = "start\n";
    int n_written;

    ptr = evbuffer_search(buf, s, strlen(s), NULL);
    if (ptr.pos == -1)
        return; /* no start string found. */

    /* Advance the pointer past the start string. */
    if (evbuffer_ptr_set(buf, &ptr, strlen(s), EVBUFFER_PTR_ADD) < 0)
        return; /* off the end of the string. */

    while (n_written < 16*1024) {
        /* Peek at a single chunk. */
```

```
        if (evbuffer_peek(buf, -1, &ptr, v, 1) < 1)
            break;
        /* Pass the data to some user-defined consume function */
        consume(v[0].iov_base, v[0].iov_len);
        n_written += v[0].iov_len;

        /* Advance the pointer so we see the next chunk next time. */
        if (evbuffer_ptr_set(buf, &ptr, v[0].iov_len, EVBUFFER_PTR_ADD)<0)
            break;
    }
}
```

NOTES

- Modifying the data pointed to by the evbuffer_iovec can result in undefined behavior.
- If any function is called that modifies the evbuffer, the pointers that evbuffer_peek() yields may become invalid.
- If your evbuffer could be used in multiple threads, make sure to lock it with evbuffer_lock() before you call evbuffer_peek(), and unlock it once you are done using the extents that evbuffer_peek() gave you.

This function is new in Libevent 2.0.2-alpha.

Adding data to an evbuffer directly

Sometimes you want to insert data info an evbuffer directly, without first writing it into a character array and then copying it in with evbuffer_add(). There are an advanced pair of functions you can use to do this: evbuffer_reserve_space() and evbuffer_commit_space(). As with evbuffer_peek(), these functions use the evbuffer_iovec structure to provide direct access to memory inside the evbuffer.

Interface

```
int evbuffer_reserve_space(struct evbuffer *buf, ev_ssize_t size,
    struct evbuffer_iovec *vec, int n_vecs);
int evbuffer_commit_space(struct evbuffer *buf,
    struct evbuffer_iovec *vec, int n_vecs);
```

The evbuffer_reserve_space() function gives you pointers to space inside the evbuffer. It expands the buffer as necessary to give you at least *size* bytes. The pointers to these extents, and their lengths, will be stored in the array of vectors you pass in with *vec*; *n_vec* is the length of this array.

The value of *n_vec* must be at least 1. If you provide only one vector, then Libevent will ensure that you have all the contiguous space you requested in a single extent, but it may have to rearrange the buffer or waste memory in order to do so. For better performance, provide at least 2 vectors. The function returns the number of provided vectors that it needed for the space you requested.

The data that you write into these vectors is not part of the buffer until you call evbuffer_commit_space(), which actually makes the data you wrote count as being in the buffer. If you want to commit less space than you asked for, you can decrease the iov_len field in any of the evbuffer_iovec structures you were given. You can also pass back fewer vectors than you were given. The evbuffer_commit_space() function returns 0 on success and -1 on failure.

NOTES AND CAVEATS

- Calling any function that rearranges the evbuffer or adds data to it evbuffer will invalidate the pointers you got from evbuffer_reserve_sp
- In the current implementation, evbuffer_reserve_space() never uses more than two vectors, no matter how many the user supplies. This may change in a future release.
- It is safe to call evbuffer_reserve_space() any number of times.

- If your evbuffer could be used in multiple threads, make sure to lock it with evbuffer_lock() before you call evbuffer_reserve_space(), and unlock it once you commit.

Example

```
/* Suppose we want to fill a buffer with 2048 bytes of output from a
   generate_data() function, without copying. */
struct evbuffer_iovec v[2];
int n, i;
size_t n_to_add = 2048;

/* Reserve 2048 bytes.*/
n = evbuffer_reserve_space(buf, n_to_add, v, 2);
if (n<=0)
   return; /* Unable to reserve the space for some reason. */

for (i=0; i<n && n_to_add > 0; ++i) {
   size_t len = v[i].iov_len;
   if (len > n_to_add) /* Don't write more than n_to_add bytes. */
      len = n_to_add;
   if (generate_data(v[i].iov_base, len) < 0) {
      /* If there was a problem during data generation, we can just stop
         here; no data will be committed to the buffer. */
      return;
   }
   /* Set iov_len to the number of bytes we actually wrote, so we
      don't commit too much. */
   v[i].iov_len = len;
}

/* We commit the space here.  Note that we give it 'i' (the number of
   vectors we actually used) rather than 'n' (the number of vectors we
   had available. */
if (evbuffer_commit_space(buf, v, i) < 0)
   return; /* Error committing */
```

Bad Examples

```
/* Here are some mistakes you can make with evbuffer_reserve().
   DO NOT IMITATE THIS CODE. */
struct evbuffer_iovec v[2];

{
  /* Do not use the pointers from evbuffer_reserve_space() after
     calling any functions that modify the buffer. */
  evbuffer_reserve_space(buf, 1024, v, 2);
  evbuffer_add(buf, "X", 1);
  /* WRONG: This next line won't work if evbuffer_add needed to rearrange
     the buffer's contents.  It might even crash your program. Instead,
     you add the data before calling evbuffer_reserve_space. */
  memset(v[0].iov_base, 'Y', v[0].iov_len-1);
  evbuffer_commit_space(buf, v, 1);
}

{
  /* Do not modify the iov_base pointers. */
  const char *data = "Here is some data";
  evbuffer_reserve_space(buf, strlen(data), v, 1);
  /* WRONG: The next line will not do what you want.  Instead, you
     should _copy_ the contents of data into v[0].iov_base. */
  v[0].iov_base = (char*) data;
  v[0].iov_len = strlen(data);
```

```
  /* In this case, evbuffer_commit_space might give an error if you're
     lucky */
  evbuffer_commit_space(buf, v, 1);
}
```

These functions have existed with their present interfaces since Libevent 2.0.2-alpha.

Network IO with evbuffers

The most common use case for evbuffers in Libevent is network IO. The interface for performing network IO on an evbuffer is:

Interface

```
int evbuffer_write(struct evbuffer *buffer, evutil_socket_t fd);
int evbuffer_write_atmost(struct evbuffer *buffer, evutil_socket_t fd,
        ev_ssize_t howmuch);
int evbuffer_read(struct evbuffer *buffer, evutil_socket_t fd, int howmuch);
```

The evbuffer_read() function reads up to *howmuch* bytes from the socket *fd* onto the end of *buffer*. It returns a number of bytes read on success, 0 on EOF, and -1 on an error. Note that the error may indicate that a nonblocking operation would not succeed; you need to check the error code for EAGAIN (or WSAEWOULDBLOCK on Windows). If *howmuch* is negative, evbuffer_read() tries to guess how much to read itself.

The evbuffer_write_atmost() function tries to write up to *howmuch* bytes from the front of *buffer* onto the socket *fd*. It returns a number of bytes written on success, and -1 on failure. As with evbuffer_read(), you need to check the error code to see whether the error is real, or just indicates that nonblocking IO could not be completed immediately. If you give a negative value for *howmuch*, we try to write the entire contents of the buffer.

Calling evbuffer_write() is the same as calling evbuffer_write_atmost() with a negative *howmuch* argument: it attempts to flush as much of the buffer as it can.

On Unix, these functions should work on any file descriptor that supports read and write. On Windows, only sockets are supported.

Note that when you are using bufferevents, you do not need to call these IO functions; the bufferevents code does it for you.

The evbuffer_write_atmost() function was introduced in Libevent 2.0.1-alpha.

Evbuffers and callbacks

Users of evbuffers frequently want to know when data is added to or removed from an evbuffer. To support this, Libevent provides a generic evbuffer callback mechanism.

Interface

```
struct evbuffer_cb_info {
        size_t orig_size;
        size_t n_added;
        size_t n_deleted;
};

typedef void (*evbuffer_cb_func)(struct evbuffer *buffer,
    const struct evbuffer_cb_info *info, void *arg);
```

An evbuffer callback is invoked whenever data is added to or removed from the evbuffer. It receives the buffer, a pointer to an evbuffer_cb_info structure, and a user-supplied argument. The evbuffer_cb_info structure's orig_size field records how many bytes there were on the buffer before its size changed; its n_added field records how many bytes were added to the buffer, and its n_deleted field records how many bytes were removed.

Interface

```
struct evbuffer_cb_entry;
struct evbuffer_cb_entry *evbuffer_add_cb(struct evbuffer *buffer,
    evbuffer_cb_func cb, void *cbarg);
```

The evbuffer_add_cb() function adds a callback to an evbuffer, and returns an opaque pointer that can later be used to refer to this particular callback instance. The *cb* argument is the function that will be invoked, and the *cbarg* is the user-supplied pointer to pass to the function.

You can have multiple callbacks set on a single evbuffer. Adding a new callback does not remove old callbacks.

Example

```
#include <event2/buffer.h>
#include <stdio.h>
#include <stdlib.h>

/* Here's a callback that remembers how many bytes we have drained in
   total from the buffer, and prints a dot every time we hit a
   megabyte. */
struct total_processed {
    size_t n;
};
void count_megabytes_cb(struct evbuffer *buffer,
    const struct evbuffer_cb_info *info, void *arg)
{
    struct total_processed *tp = arg;
    size_t old_n = tp->n;
    int megabytes, i;
    tp->n += info->n_deleted;
    megabytes = ((tp->n) >> 20) - (old_n >> 20);
    for (i=0; i<megabytes; ++i)
        putc('.', stdout);
}

void operation_with_counted_bytes(void)
{
    struct total_processed *tp = malloc(sizeof(*tp));
    struct evbuffer *buf = evbuffer_new();
    tp->n = 0;
    evbuffer_add_cb(buf, count_megabytes_cb, tp);

    /* Use the evbuffer for a while.  When we're done: */
    evbuffer_free(buf);
    free(tp);
}
```

Note in passing that freeing a nonempty evbuffer does not count as draining data from it, and that freeing an evbuffer does not free the user-supplied data pointer for its callbacks.

If you don't want a callback to be permanently active on a buffer, you can *remove* it (to make it gone for good), or disable it (to turn it off for a while):

Interface

```
int evbuffer_remove_cb_entry(struct evbuffer *buffer,
    struct evbuffer_cb_entry *ent);
int evbuffer_remove_cb(struct evbuffer *buffer, evbuffer_cb_func cb,
    void *cbarg);

#define EVBUFFER_CB_ENABLED 1
int evbuffer_cb_set_flags(struct evbuffer *buffer,
                          struct evbuffer_cb_entry *cb,
```

```
                            ev_uint32_t flags);
int evbuffer_cb_clear_flags(struct evbuffer *buffer,
                            struct evbuffer_cb_entry *cb,
                            ev_uint32_t flags);
```

You can remove a callback either by the evbuffer_cb_entry you got when you added it, or by the callback and pointer you used. The evbuffer_remove_cb() functions return 0 on success and -1 on failure.

The evbuffer_cb_set_flags() function and the evbuffer_cb_clear_flags() function make a given flag be set or cleared on a given callback respectively. Right now, only one user-visible flag is supported: *EVBUFFER_CB_ENABLED*. The flag is set by default. When it is cleared, modifications to the evbuffer do not cause this callback to get invoked.

Interface

```
int evbuffer_defer_callbacks(struct evbuffer *buffer, struct event_base *base);
```

As with bufferevent callbacks, you can cause evbuffer callbacks to not run immediately when the evbuffer is changed, but rather to be *deferred* and run as part of the event loop of a given event base. This can be helpful if you have multiple evbuffers whose callbacks potentially cause data to be added and removed from one another, and you want to avoid smashing the stack.

If an evbuffer's callbacks are deferred, then when they are finally invoked, they may summarize the results for multiple operations.

Like bufferevents, evbuffers are internally reference-counted, so that it is safe to free an evbuffer even if it has deferred callbacks that have not yet executed.

This entire callback system was new in Libevent 2.0.1-alpha. The evbuffer_cb_(setlclear)_flags() functions have existed with their present interfaces since 2.0.2-alpha.

Avoiding data copies with evbuffer-based IO

Really fast network programming often calls for doing as few data copies as possible. Libevent provides some mechanisms to help out with this.

Interface

```
typedef void (*evbuffer_ref_cleanup_cb)(const void *data,
    size_t datalen, void *extra);

int evbuffer_add_reference(struct evbuffer *outbuf,
    const void *data, size_t datlen,
    evbuffer_ref_cleanup_cb cleanupfn, void *extra);
```

This function adds a piece of data to the end of an evbuffer by reference. No copy is performed: instead, the evbuffer just stores a pointer to the *datlen* bytes stored at *data*. Therefore, the pointer must remain valid for as long as the evbuffer is using it. When the evbuffer no longer needs data, it will call the provided "cleanupfn" function with the provided "data" pointer, "datlen" value, and "extra" pointer as arguments. This function returns 0 on success, -1 on failure.

Example

```
#include <event2/buffer.h>
#include <stdlib.h>
#include <string.h>

/* In this example, we have a bunch of evbuffers that we want to use to
   spool a one-megabyte resource out to the network.  We do this
   without keeping any more copies of the resource in memory than
   necessary. */

#define HUGE_RESOURCE_SIZE (1024*1024)
struct huge_resource {
    /* We keep a count of the references that exist to this structure,
       so that we know when we can free it. */
```

```
    int reference_count;
    char data[HUGE_RESOURCE_SIZE];
};

struct huge_resource *new_resource(void) {
    struct huge_resource *hr = malloc(sizeof(struct huge_resource));
    hr->reference_count = 1;
    /* Here we should fill hr->data with something.  In real life,
       we'd probably load something or do a complex calculation.
       Here, we'll just fill it with EEs. */
    memset(hr->data, 0xEE, sizeof(hr->data));
    return hr;
}

void free_resource(struct huge_resource *hr) {
    --hr->reference_count;
    if (hr->reference_count == 0)
        free(hr);
}

static void cleanup(const void *data, size_t len, void *arg) {
    free_resource(arg);
}

/* This is the function that actually adds the resource to the
   buffer. */
void spool_resource_to_evbuffer(struct evbuffer *buf,
    struct huge_resource *hr)
{
    ++hr->reference_count;
    evbuffer_add_reference(buf, hr->data, HUGE_RESOURCE_SIZE,
        cleanup, hr);
}
```

The evbuffer_add_reference() function has had is present interface since 2.0.2-alpha.

Adding a file to an evbuffer

Some operating systems provide ways to write files to the network without ever copying the data to userspace. You can access these mechanisms, where available, with the simple interface:

Interface

```
int evbuffer_add_file(struct evbuffer *output, int fd, ev_off_t offset,
    size_t length);
```

The evbuffer_add_file() function assumes that it has an open file descriptor (not a socket, for once!) *fd* that is available for reading. It adds *length* bytes from the file, starting at position *offset*, to the end of *output*. It returns 0 on success, or -1 on failure.

WARNING In Libevent 2.0.x, the only reliable thing to do with data added this way was to send it to the network with evbuffer_write*(), drain it with evbuffer_drain(), or move it to another evbuffer with evbuffer_*_buffer(). You couldn't reliably extract it from the buffer with evbuffer_remove(), linearize it with evbuffer_pullup(), and so on. Libevent 2.1.x tries to fix this limitation.

If your operating system supports splice() or sendfile(), Libevent will use it to send data from *fd* to the network directly when call evbuffer_write(), without copying the data to user RAM at all. If splice/sendfile don't exist, but you have mmap(), Libevent will mmap the file, and your kernel can hopefully figure out that it never needs to copy the data to userspace. Otherwise, Libevent will just read the data from disk into RAM.

The file descriptor will be closed after the data is flushed from the evbuffer, or when the evbuffer is freed. If that's not what you want, or if you want finer-grained control over the file, see the file_segment functionality below.

This function was introduced in Libevent 2.0.1-alpha.

Fine-grained control with file segments

The evbuffer_add_file() interface is inefficient for adding the same file more than once, since it takes ownership of the file.

Interface

```
struct evbuffer_file_segment;

struct evbuffer_file_segment *evbuffer_file_segment_new(
        int fd, ev_off_t offset, ev_off_t length, unsigned flags);
void evbuffer_file_segment_free(struct evbuffer_file_segment *seg);
int evbuffer_add_file_segment(struct evbuffer *buf,
    struct evbuffer_file_segment *seg, ev_off_t offset, ev_off_t length);
```

The evbuffer_file_segment_new() function creates and returns a new evbuffer_file_segment object to represent a piece of the underlying file stored in *fd* that begins at *offset* and contains *length* bytes. On error, it return NULL.

File segments are implemented with sendfile, splice, mmap, CreateFileMapping, or malloc()-and-read(), as appropriate. They're created using the most lightweight supported mechanism, and transition to a heavier-weight mechanism as needed. (For example, if your OS supports sendfile and mmap, then a file segment can be implemented using only sendfile, until you try to actually inspect its contents. At that point, it needs to be mmap()ed.) You can control the fine-grained behavior of a file segment with these flags:

EVBUF_FS_CLOSE_ON_FREE
: If this flag is set, freeing the file segment with evbuffer_file_segment_free() will close the underlying file.

EVBUF_FS_DISABLE_MMAP
: If this flag is set, the file_segment will never use a mapped-memory style backend (CreateFileMapping, mmap) for this file, even if that would be appropriate.

EVBUF_FS_DISABLE_SENDFILE
: If this flag is set, the file_segment will never use a sendfile-style backend (sendfile, splice) for this file, even if that would be appropriate.

EVBUF_FS_DISABLE_LOCKING
: If this flag is set, no locks are allocated for the file segment: it won't be safe to use it in any way where it can be seen by multiple threads.

Once you have an evbuffer_file_segment, you can add some or all of it to an evbuffer using evbuffer_add_file_segment(). The *offset* argument here refers to an offset within the file segment, not to an offset within the file itself.

When you no longer want to use a file segment, you can free it with evbuffer_file_segment_free(). The actual storage won't be released until no evbuffer any longer holds a reference to a piece of the file segment.

Interface

```
typedef void (*evbuffer_file_segment_cleanup_cb)(
    struct evbuffer_file_segment const *seg, int flags, void *arg);

void evbuffer_file_segment_add_cleanup_cb(struct evbuffer_file_segment *seg,
        evbuffer_file_segment_cleanup_cb cb, void *arg);
```

You can add a callback function to a file segment that will be invoked when the final reference to the file segment has been released and the file segment is about to get freed. This callback **must not** attempt to revivify the file segment, add it to any buffers, or so on.

These file-segment functions first appeared in Libevent 2.1.1-alpha; evbuffer_file_segment_add_cleanup_cb() was added in 2.1.2-alpha.

Adding an evbuffer to another by reference

You can also add one evbuffer's to another by reference: rather than removing the contents of one buffer and adding them to another, you give one evbuffer a reference to another, and it behaves as though you had copied all the bytes in.

Interface

```
int evbuffer_add_buffer_reference(struct evbuffer *outbuf,
    struct evbuffer *inbuf);
```

The evbuffer_add_buffer_reference() function behaves as though you had copied all the data from *outbuf* to *inbuf*, but does not perform any unnecessary copies. It returns 0 if successful and -1 on failure.

Note that subsequent changes to the contents of *inbuf* are not reflected in *outbuf*: this function adds the current contents of the evbuffer by reference, not the evbuffer itself.

Note also that you cannot nest buffer references: a buffer that has already been the *outbuf* of one evbuffer_add_buffer_reference call cannot be the *inbuf* of another.

This function was introduced in Libevent 2.1.1-alpha.

Making an evbuffer add- or remove-only

Interface

```
int evbuffer_freeze(struct evbuffer *buf, int at_front);
int evbuffer_unfreeze(struct evbuffer *buf, int at_front);
```

You can use these functions to temporarily disable changes to the front or end of an evbuffer. The bufferevent code uses them internally to prevent accidental modifications to the front of an output buffer, or the end of an input buffer.

The evbuffer_freeze() functions were introduced in Libevent 2.0.1-alpha.

Obsolete evbuffer functions

The evbuffer interface changed a lot in Libevent 2.0. Before then, every evbuffers was implemented as a contiguous chunk of RAM, which made access very inefficient.

The event.h header used to expose the internals of struct evbuffer. These are no longer available; they changed too much between 1.4 and 2.0 for any code that relied on them to work.

To access the number of bytes in an evbuffer, there was an EVBUFFER_LENGTH() macro. The actual data was available with EVBUFFER_DATA(). These are both available in event2/buffer_compat.h. Watch out, though: EVBUFFER_DATA(b) is an alias for evbuffer_pullup(b, -1), which can be very expensive.

Some other deprecated interfaces are:

Deprecated Interface

```
char *evbuffer_readline(struct evbuffer *buffer);
unsigned char *evbuffer_find(struct evbuffer *buffer,
    const unsigned char *what, size_t len);
```

The evbuffer_readline() function worked like the current evbuffer_readln(buffer, NULL, EVBUFFER_EOL_ANY).

The evbuffer_find() function would search for the first occurrence of a string in a buffer, and return a pointer to it. Unlike evbuffer_search(), it could only find the first string. To stay compatible with old code that uses this function, it now linearizes the entire buffer up to the end of the located string.

The callback interface was different too:

Deprecated Interface

```
typedef void (*evbuffer_cb)(struct evbuffer *buffer,
    size_t old_len, size_t new_len, void *arg);
void evbuffer_setcb(struct evbuffer *buffer, evbuffer_cb cb, void *cbarg);
```

An evbuffer could only have one callback set at a time, so setting a new callback would disable the previous callback, and setting a callback of NULL was the preferred way to disable a callbacks.

Instead of getting an evbuffer_cb_info_structure, the function was called with the old and new lengths of the evbuffer. Thus, if old_len was greater than new_len, data was drained. If new_len was greater than old_len, data was added. It was not possible to defer callbacks, and so adds and deletes were never batched into a single callback invocation.

The obsolete functions here are still available in event2/buffer_compat.h.

Connection listeners: accepting TCP connections

The evconnlistener mechanism gives you a way to listen for and accept incoming TCP connections.

All the functions and types in this section are declared in event2/listener.h. They first appeared in Libevent 2.0.2-alpha, unless otherwise noted.

Creating or freeing an evconnlistener

Interface

```
struct evconnlistener *evconnlistener_new(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    evutil_socket_t fd);
struct evconnlistener *evconnlistener_new_bind(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    const struct sockaddr *sa, int socklen);
void evconnlistener_free(struct evconnlistener *lev);
```

The two evconnlistener_new*() functions both allocate and return a new connection listener object. A connection listener uses an event_base to note when there is a new TCP connection on a given listener socket. When a new connection arrives, it invokes the callback function you give it.

In both functions, the *base* parameter is an event_base that the listener should use to listen for connections. The *cb* function is a callback to invoke when a new connection is received; if *cb* is NULL, the listener is treated as disabled until a callback is set. The *ptr* pointer will be passed to the callback. The *flags* argument controls the behavior of the listener — more on this below. The *backlog* parameter controls the maximum number of pending connections that the network stack should allow to wait in a not-yet-accepted state at any time; see documentation for your system's listen() function for more details. If *backlog* is negative, Libevent tries to pick a good value for the backlog; if it is zero, Libevent assumes that you have already called listen() on the socket you are providing it.

The functions differ in how they set up their listener socket. The evconnlistener_new() function assumes that you have already bound a socket to the port you want to listen on, and that you're passing the socket in as *fd*. If you want Libevent to allocate and bind to a socket on its own, call evconnlistener_new_bind(), and pass in the sockaddr you want to bind to, and its length.

Tip
[When using evconnlistener_new, make sure your listening socket is in non-blocking mode by using evutil_make_socket_nonblocking or by manually setting the correct socket option. When the listening socket is left in blocking mode, undefined behavior might occur.]

To free a connection listener, pass it to evconnlistener_free().

Recognized flags

These are the flags you can pass to the *flags* argument of the evconnlistener_new() function. You can give any number of these, OR'd together.

LEV_OPT_LEAVE_SOCKETS_BLOCKING
By default, when the connection listener accepts a new incoming socket, it sets it up to be nonblocking so that you can use it with the rest of Libevent. Set this flag if you do not want this behavior.

LEV_OPT_CLOSE_ON_FREE
If this option is set, the connection listener closes its underlying socket when you free it.

LEV_OPT_CLOSE_ON_EXEC
If this option is set, the connection listener sets the close-on-exec flag on the underlying listener socket. See your platform documentation for fcntl and FD_CLOEXEC for more information.

LEV_OPT_REUSEABLE
By default on some platforms, once a listener socket is closed, no other socket can bind to the same port until a while has passed. Setting this option makes Libevent mark the socket as reusable, so that once it is closed, another socket can be opened to listen on the same port.

LEV_OPT_THREADSAFE
Allocate locks for the listener, so that it's safe to use it from multiple threads. New in Libevent 2.0.8-rc.

LEV_OPT_DISABLED
Initialize the listener to be disabled, not enabled. You can turn it on manually with evconnlistener_enable(). New in Libevent 2.1.1-alpha.

LEV_OPT_DEFERRED_ACCEPT
If possible, tell the kernel to not announce sockets as having been accepted until some data has been received on them, and they are ready for reading. Do not use this option if your protocol *doesn't* start out with the client transmitting data, since in that case this option will sometimes cause the kernel to never tell you about the connection. Not all operating systems support this option: on ones that don't, this option has no effect. New in Libevent 2.1.1-alpha.

The connection listener callback

Interface

```
typedef void (*evconnlistener_cb)(struct evconnlistener *listener,
    evutil_socket_t sock, struct sockaddr *addr, int len, void *ptr);
```

When a new connection is received, the provided callback function is invoked. The *listener* argument is the connection listener that received the connection. The *sock* argument is the new socket itself. The *addr* and *len* arguments are the address from which the connection was received and the length of that address respectively. The *ptr* argument is the user-supplied pointer that was passed to evconnlistener_new().

Enabling and disabling an evconnlistener

Interface

```
int evconnlistener_disable(struct evconnlistener *lev);
int evconnlistener_enable(struct evconnlistener *lev);
```

These functions temporarily disable or reenable listening for new connections.

Adjusting an evconnlistener's callback

Interface

```
void evconnlistener_set_cb(struct evconnlistener *lev,
    evconnlistener_cb cb, void *arg);
```

This function adjusts the callback and callback argument of an existing evconnlistener. It was introduced in 2.0.9-rc.

Inspecting an evconnlistener

Interface

```
evutil_socket_t evconnlistener_get_fd(struct evconnlistener *lev);
struct event_base *evconnlistener_get_base(struct evconnlistener *lev);
```

These functions return a listener's associated socket and event_base respectively.

The evconnlistener_get_fd() function first appeared in Libevent 2.0.3-alpha.

Detecting errors

You can set an error callback that gets informed whenever an accept() call fails on the listener. This can be important to do if you're facing an error condition that would lock the process unless you addressed it.

Interface

```
typedef void (*evconnlistener_errorcb)(struct evconnlistener *lis, void *ptr);
void evconnlistener_set_error_cb(struct evconnlistener *lev,
    evconnlistener_errorcb errorcb);
```

If you use evconnlistener_set_error_cb() to set an error callback on a listener, the callback will be invoked every time that an error occurs on the listener. It will receive the listener as its first argument, and the argument passed as *ptr* to evconnlistener_new() as its second argument.

This function was introduced in Libevent 2.0.8-rc.

Example code: an echo server.

Example

```
#include <event2/listener.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>

#include <arpa/inet.h>

#include <string.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include <errno.h>

static void
echo_read_cb(struct bufferevent *bev, void *ctx)
{
        /* This callback is invoked when there is data to read on bev. */
        struct evbuffer *input = bufferevent_get_input(bev);
        struct evbuffer *output = bufferevent_get_output(bev);

        /* Copy all the data from the input buffer to the output buffer. */
        evbuffer_add_buffer(output, input);
}

static void
echo_event_cb(struct bufferevent *bev, short events, void *ctx)
{
        if (events & BEV_EVENT_ERROR)
                perror("Error from bufferevent");
        if (events & (BEV_EVENT_EOF | BEV_EVENT_ERROR)) {
                bufferevent_free(bev);
        }
}

static void
accept_conn_cb(struct evconnlistener *listener,
    evutil_socket_t fd, struct sockaddr *address, int socklen,
    void *ctx)
{
        /* We got a new connection! Set up a bufferevent for it. */
        struct event_base *base = evconnlistener_get_base(listener);
        struct bufferevent *bev = bufferevent_socket_new(
                base, fd, BEV_OPT_CLOSE_ON_FREE);

        bufferevent_setcb(bev, echo_read_cb, NULL, echo_event_cb, NULL);

        bufferevent_enable(bev, EV_READ|EV_WRITE);
}

static void
accept_error_cb(struct evconnlistener *listener, void *ctx)
{
        struct event_base *base = evconnlistener_get_base(listener);
        int err = EVUTIL_SOCKET_ERROR();
        fprintf(stderr, "Got an error %d (%s) on the listener. "
                "Shutting down.\n", err, evutil_socket_error_to_string(err));

        event_base_loopexit(base, NULL);
}

int
main(int argc, char **argv)
{
        struct event_base *base;
        struct evconnlistener *listener;
        struct sockaddr_in sin;

        int port = 9876;

        if (argc > 1) {
                port = atoi(argv[1]);
        }
```

```
        if (port<=0 || port>65535) {
                puts("Invalid port");
                return 1;
        }

        base = event_base_new();
        if (!base) {
                puts("Couldn't open event base");
                return 1;
        }

        /* Clear the sockaddr before using it, in case there are extra
         * platform-specific fields that can mess us up. */
        memset(&sin, 0, sizeof(sin));
        /* This is an INET address */
        sin.sin_family = AF_INET;
        /* Listen on 0.0.0.0 */
        sin.sin_addr.s_addr = htonl(0);
        /* Listen on the given port. */
        sin.sin_port = htons(port);

        listener = evconnlistener_new_bind(base, accept_conn_cb, NULL,
            LEV_OPT_CLOSE_ON_FREE|LEV_OPT_REUSEABLE, -1,
            (struct sockaddr*)&sin, sizeof(sin));
        if (!listener) {
                perror("Couldn't create listener");
                return 1;
        }
        evconnlistener_set_error_cb(listener, accept_error_cb);

        event_base_dispatch(base);
        return 0;
}
```


Using DNS with Libevent: high and low-level functionality

Libevent provides a few APIs to use for resolving DNS names, and a facility for implementing simple DNS servers.

We'll start by describing the higher-level facilities for name lookup, and then describe the low-level and server facilities.

Note There are known limitations in Libevent's current DNS client implementation. It doesn't support TCP lookups, DNSSec, or arbitrary record types. We'd like to fix all of these in some future version of Libevent, but for now, they're not there.

Preliminaries: Portable blocking name resolution

To aid in porting programs that already use blocking name resolution, Libevent provides a portable implementation of the standard getaddrinfo() interface. This can be helpful when your program needs to run on platforms where either there is no getaddrinfo()

function, or where getaddrinfo() doesn't conform to the standard as well as our replacement. (There are shockingly many of each.)

The getaddrinfo() interface is specified in RFC 3493, section 6.1. See the "Compatibility Notes" section below for a summary of how we fall short of a conformant implemenation.

Interface

```
struct evutil_addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    size_t ai_addrlen;
    char *ai_canonname;
    struct sockaddr *ai_addr;
    struct evutil_addrinfo *ai_next;
};

#define EVUTIL_AI_PASSIVE     /* ... */
#define EVUTIL_AI_CANONNAME   /* ... */
#define EVUTIL_AI_NUMERICHOST /* ... */
#define EVUTIL_AI_NUMERICSERV /* ... */
#define EVUTIL_AI_V4MAPPED    /* ... */
#define EVUTIL_AI_ALL         /* ... */
#define EVUTIL_AI_ADDRCONFIG  /* ... */

int evutil_getaddrinfo(const char *nodename, const char *servname,
    const struct evutil_addrinfo *hints, struct evutil_addrinfo **res);
void evutil_freeaddrinfo(struct evutil_addrinfo *ai);
const char *evutil_gai_strerror(int err);
```

The evutil_getaddrinfo() function tries to resolve the provided nodename and servname fields, according to the rules you give it in *hints*, and build you a linked list of evutil_addrinfo structures and store them in *res. It returns 0 on success, and a nonzero error code on failure.

You must provide at least one of *nodename* and *servname*. If *nodename* is provided, it is either a literal IPv4 address (like "127.0.0.1"), a literal IPv6 address (like "::1"), or a DNS name (like "www.example.com"). If *servname* is provided, it is either the symbolic name of a network service (like "https") or a string containing a port number given in decimal (like "443").

If you do not specify *servname*, then the port values in *res will be set to zero. If you do not specify *nodename*, then the addresses in *res will either be for localhost (by default), or for "any" (if EVUTIL_AI_PASSIVE is set.)

The ai_flags field of *hints* tells evutil_getaddrinfo how to perform the lookup. It can contain zero or more of the flags below, ORed together.

EVUTIL_AI_PASSIVE
: This flag indicates that we're going to be using the address for listening, not for connection. Ordinarily this makes no difference, except when *nodename* is NULL: for connecting, a NULL nodename is localhost (127.0.0.1 or ::1), whereas when listening, a NULL node name is ANY (0.0.0.0 or ::0).

EVUTIL_AI_CANONNAME
: If this flag is set, we try to report the canonical name for the host in the ai_canonname field.

EVUTIL_AI_NUMERICHOST
: When this flag is set, we only resolve numeric IPv4 and IPv6 addresses; if the *nodename* would require a name lookup, we instead give an EVUTIL_EAI_NONAME error.

EVUTIL_AI_NUMERICSERV
: When this flag is set, we only resolve numeric service names. If the *servname* is neither NULL nor a decimal integer, give an EVUTIL_EAI_NONAME error.

EVUTIL_AI_V4MAPPED
This flag indicates that if ai_family is AF_INET6, and no IPv6 addresses are found, any IPv4 addresses in the result should be returned as v4-mapped IPv6 addresses. It is not currently supported by evutil_getaddrinfo() unless the OS supports it.

EVUTIL_AI_ALL
If this flag and EVUTIL_AI_V4MAPPED are both set, then IPv4 addresses in the result included in the result as 4-mapped IPv6 addresses, whether there are any IPv6 addresses or not. It is not currently supported by evutil_getaddrinfo() unless the OS supports it.

EVUTIL_AI_ADDRCONFIG
If this flag is set, then IPv4 addresses are only included in the result if the system has a nonlocal IPv4 address, and IPv6 addresses are only included in the result if the system has a nonlocal IPv6 address.

The ai_family field of *hints* is used to tell evutil_getaddrinfo() which addresses it should return. It can be AF_INET to request IPv4 addresses only, AF_INET6 to request IPv6 addresses only, or AF_UNSPEC to request all available addresses.

The ai_socktype and ai_protocol fields of *hints* are used to tell evutil_getaddrinfo() how you're going to use the address. They're the same as the socktype and protocol fields you would pass to socket().

If evutil_getaddrinfo() is successful, it allocates a new linked list of evutil_addrinfo structures, where each points to the next with its "ai_next" pointer, and stores them in *res. Because this value is heap-allocated, you will need to use evutil_freeaddrinfo to free it.

If it fails, it returns one of these numeric error codes:

EVUTIL_EAI_ADDRFAMILY
You requested an address family that made no sense for the nodename.

EVUTIL_EAI_AGAIN
There was a recoverable error in name resolution; try again later.

EVUTIL_EAI_FAIL
There was a non-recoverable error in name resolution; your resolver or your DNS server may be busted.

EVUTIL_EAI_BADFLAGS
The ai_flags field in hints was somehow invalid.

EVUTIL_EAI_FAMILY
The ai_family field in hints was not one we support.

EVUTIL_EAI_MEMORY
We ran out of memory while trying to answer your request.

EVUTIL_EAI_NODATA
The host you asked for exists, but has no address information associated with it. (Or, it has no address information of the type you requested.)

EVUTIL_EAI_NONAME
The host you asked for doesn't seem to exist.

EVUTIL_EAI_SERVICE
The service you asked for doesn't seem to exist.

EVUTIL_EAI_SOCKTYPE
We don't support the socket type you asked for, or it isn't compatible with ai_protocol.

EVUTIL_EAI_SYSTEM
There was some other system error during name resolution. Check errno for more information.

EVUTIL_EAI_CANCEL
The application requested that this DNS lookup should be canceled before it was finished. The evutil_getaddrinfo() function never produces this error, but it can come from evdns_getaddrinfo() as described in the section below.

You can use evutil_gai_strerror() to convert one of these results into a human-readable string.

Note: If your OS defines struct addrinfo, then evutil_addrinfo is just an alias for your OS's built-in structure. Similarly, if your operating system defines any of the AI_* flags, then the corresponding EVUTIL_AI_* flag is just an alias for the native flag; and if your operating system defines any of the EAI_* errors, then the corresponding EVUTIL_EAI_* code is the same as your platform's native error code.

Example: Resolving a hostname and making a blocking connection

```
#include <event2/util.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <unistd.h>

evutil_socket_t
get_tcp_socket_for_host(const char *hostname, ev_uint16_t port)
{
    char port_buf[6];
    struct evutil_addrinfo hints;
    struct evutil_addrinfo *answer = NULL;
    int err;
    evutil_socket_t sock;

    /* Convert the port to decimal. */
    evutil_snprintf(port_buf, sizeof(port_buf), "%d", (int)port);

    /* Build the hints to tell getaddrinfo how to act. */
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC; /* v4 or v6 is fine. */
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP; /* We want a TCP socket */
    /* Only return addresses we can use. */
    hints.ai_flags = EVUTIL_AI_ADDRCONFIG;

    /* Look up the hostname. */
    err = evutil_getaddrinfo(hostname, port_buf, &hints, &answer);
    if (err != 0) {
          fprintf(stderr, "Error while resolving '%s': %s",
                  hostname, evutil_gai_strerror(err));
          return -1;
    }

    /* If there was no error, we should have at least one answer. */
    assert(answer);
    /* Just use the first answer. */
    sock = socket(answer->ai_family,
                  answer->ai_socktype,
                  answer->ai_protocol);
    if (sock < 0)
        return -1;
    if (connect(sock, answer->ai_addr, answer->ai_addrlen)) {
        /* Note that we're doing a blocking connect in this function.
         * If this were nonblocking, we'd need to treat some errors
         * (like EINTR and EAGAIN) specially. */
        EVUTIL_CLOSESOCKET(sock);
        return -1;
    }

    return sock;
```

```
}
```

These functions and constants were new in Libevent 2.0.3-alpha. They are declared in event2/util.h.

Non-blocking hostname resolution with evdns_getaddrinfo()

The main problem with the regular getaddrinfo() interface, and with evutil_getaddrinfo() above, is that they're blocking: when you call them, the thread you're in has to wait while they query your DNS server(s) and wait for a response. Since you're using Libevent, that probably isn't the behavior you want.

So for nonblocking use, Libevent provides a set of functions to launch DNS requests, and use Libevent to wait for the server to answer.

Interface

```
typedef void (*evdns_getaddrinfo_cb)(
    int result, struct evutil_addrinfo *res, void *arg);
struct evdns_getaddrinfo_request;

struct evdns_getaddrinfo_request *evdns_getaddrinfo(
    struct evdns_base *dns_base,
    const char *nodename, const char *servname,
    const struct evutil_addrinfo *hints_in,
    evdns_getaddrinfo_cb cb, void *arg);

void evdns_getaddrinfo_cancel(struct evdns_getaddrinfo_request *req);
```

The evdns_getaddrinfo() function behaves just like evutil_getaddrinfo(), except that instead of blocking on DNS servers, it uses Libevent's low-level DNS facilities to look hostnames up for you. Because it can't always return you the result immediately, you need to provide it a callback function of type evdns_getaddrinfo_cb, and an optional user-supplied argument for that callback function.

Additionally, you need to provide evdns_getaddrinfo() with a pointer to an evdns_base. This structure holds the state and configuration for Libevent's DNS resolver. See the next section for more information on how to get one.

The evdns_getaddrinfo() function returns NULL if it fails or succeeds immediately. Otherwise, it returns a pointer to an evdns_getaddrinfo_request. You can use this to cancel the request with evdns_getaddrinfo_cancel() at any time before the request is finished.

Note that the callback function *will* eventually be invoked whether evdns_getaddrinfo() returns NULL or not, and whether evdns_getaddrinfo_cancel() is called or not.

When you call evdns_getaddrinfo(), it makes its own internal copies of its nodename, servname, and hints arguments: you do not need to ensure that they continue to exist while the name lookup is in progress.

Example: Nonblocking lookups with evdns_getaddrinfo()

```
#include <event2/dns.h>
#include <event2/util.h>
#include <event2/event.h>

#include <sys/socket.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

int n_pending_requests = 0;
struct event_base *base = NULL;

struct user_data {
```

```
    char *name; /* the name we're resolving */
    int idx; /* its position on the command line */
};

void callback(int errcode, struct evutil_addrinfo *addr, void *ptr)
{
    struct user_data *data = ptr;
    const char *name = data->name;
    if (errcode) {
        printf("%d. %s -> %s\n", data->idx, name, evutil_gai_strerror(errcode));
    } else {
        struct evutil_addrinfo *ai;
        printf("%d. %s", data->idx, name);
        if (addr->ai_canonname)
            printf(" [%s]", addr->ai_canonname);
        puts("");
        for (ai = addr; ai; ai = ai->ai_next) {
            char buf[128];
            const char *s = NULL;
            if (ai->ai_family == AF_INET) {
                struct sockaddr_in *sin = (struct sockaddr_in *)ai->ai_addr;
                s = evutil_inet_ntop(AF_INET, &sin->sin_addr, buf, 128);
            } else if (ai->ai_family == AF_INET6) {
                struct sockaddr_in6 *sin6 = (struct sockaddr_in6 *)ai->ai_addr;
                s = evutil_inet_ntop(AF_INET6, &sin6->sin6_addr, buf, 128);
            }
            if (s)
                printf("    -> %s\n", s);
        }
        evutil_freeaddrinfo(addr);
    }
    free(data->name);
    free(data);
    if (--n_pending_requests == 0)
        event_base_loopexit(base, NULL);
}

/* Take a list of domain names from the command line and resolve them in
 * parallel. */
int main(int argc, char **argv)
{
    int i;
    struct evdns_base *dnsbase;

    if (argc == 1) {
        puts("No addresses given.");
        return 0;
    }
    base = event_base_new();
    if (!base)
        return 1;
    dnsbase = evdns_base_new(base, 1);
    if (!dnsbase)
        return 2;

    for (i = 1; i < argc; ++i) {
        struct evutil_addrinfo hints;
        struct evdns_getaddrinfo_request *req;
        struct user_data *user_data;
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = AF_UNSPEC;
        hints.ai_flags = EVUTIL_AI_CANONNAME;
```

```
        /* Unless we specify a socktype, we'll get at least two entries for
         * each address: one for TCP and one for UDP. That's not what we
         * want. */
        hints.ai_socktype = SOCK_STREAM;
        hints.ai_protocol = IPPROTO_TCP;

        if (!(user_data = malloc(sizeof(struct user_data)))) {
            perror("malloc");
            exit(1);
        }
        if (!(user_data->name = strdup(argv[i]))) {
            perror("strdup");
            exit(1);
        }
        user_data->idx = i;

        ++n_pending_requests;
        req = evdns_getaddrinfo(
                         dnsbase, argv[i], NULL /* no service name given */,
                         &hints, callback, user_data);
        if (req == NULL) {
          printf("    [request for %s returned immediately]\n", argv[i]);
          /* No need to free user_data or decrement n_pending_requests; that
           * happened in the callback. */
        }
    }

    if (n_pending_requests)
      event_base_dispatch(base);

    evdns_base_free(dnsbase, 0);
    event_base_free(base);

    return 0;
}
```

These functions were new in Libevent 2.0.3-alpha. They are declared in event2/dns.h.

Creating and configuring an evdns_base

Before you can do nonblocking DNS lookups with evdns, you'll need to configure an evdns_base. Each evdns_base stores a list of nameservers, and DNS configuration options, and tracks active and in-flight DNS requests.

Interface

```
struct evdns_base *evdns_base_new(struct event_base *event_base,
       int initialize);
void evdns_base_free(struct evdns_base *base, int fail_requests);
```

The evdns_base_new() function returns a new evdns_base on success, and NULL on failure. If the *initialize* argument is 1, it tries to configure the DNS base sensibly given your operating system's default. If it is 0, it leaves the evdns_base empty, with no nameservers or options configured.

When you no longer need an evdns_base, you can free it with evdns_base_free. If its *fail_requests* argument is true, it will make all in-flight requests get their callbacks invoked with a *canceled* error code before it frees the base.

Initializing evdns from the system configuration

If you want a little more control over how the evdns_base is initialized, you can pass 0 as the *initialize* argument to evdns_base_new, and invoke one of these functions.

Interface

```
#define DNS_OPTION_SEARCH 1
#define DNS_OPTION_NAMESERVERS 2
#define DNS_OPTION_MISC 4
#define DNS_OPTION_HOSTSFILE 8
#define DNS_OPTIONS_ALL 15
int evdns_base_resolv_conf_parse(struct evdns_base *base, int flags,
                                 const char *filename);

#ifdef WIN32
int evdns_base_config_windows_nameservers(struct evdns_base *);
#define EVDNS_BASE_CONFIG_WINDOWS_NAMESERVERS_IMPLEMENTED
#endif
```

The evdns_base_resolv_conf_parse() function will scan the resolv.conf formatted file stored in *filename*, and read in all the options from it that are listed in *flags*. (For more information on the resolv.conf file, see your local Unix manual pages.)

DNS_OPTION_SEARCH
Tells evdns to read the *domain* and *search* fields from the resolv.conf file and the *ndots* option, and use them to decide which domains (if any) to search for hostnames that aren't fully-qualified.

DNS_OPTION_NAMESERVERS
This flag tells evdns to learn the nameservers from the resolv.conf file.

DNS_OPTION_MISC
Tells evdns to set other configuration options from the resolv.conf file.

DNS_OPTION_HOSTSFILE
Tells evdns to read a list of hosts from /etc/hosts as part of loading the resolv.conf file.

DNS_OPTIONS_ALL
Tells evdns to learn as much as it can from the resolv.conf file.

On Windows, you don't have a resolv.conf file to tell you where your nameservers are, so you can use the evdns_base_config_windows_na function to read all your nameservers from your registry (or your NetworkParams, or wherever they're hidden).

The resolv.conf file format

The resolv.conf format we recognize is a text file, each line of which should either be empty, contain a comment starting with the # character, or consist of a token followed zero or more arguments. The tokens we recognize are:

nameserver
Must be followed by the IP address of exactly one nameserver. As an extension, Libevent allows you to specify a nonstandard port for the nameserver, using the IP:Port or the [IPv6]:port syntax.

domain
The local domain name.

search
A list of names to search when resolving local hostnames. Any name that has fewer than "ndots" dots in it is considered local, and if we can't resolve it as-is, we look in these domain names. For example, if "search" is example.com and "ndots" is 1, then when the user asks us to resolve "www", we will consider "www.example.com".

options
A space-separated list of options. Each option is given either as a bare string, or (if it takes an argument) in the option:value format. Recognized options are:

ndots:INTEGER
Used to configure searching. See "search" above. Defaults to 1.

timeout:FLOAT
How long, in seconds, do we wait for a response from a DNS server before we assume we aren't getting one? Defaults to 5 seconds.

max-timeouts:INT
How many times do we allow a nameserver to time-out in a row before we assume that it's down? Defaults to 3.

max-inflight:INT
How many DNS requests do we allow to be pending at once? (If we try to do more requests than this, the extras will stall until the earlier ones are answered or time out.) Defaults to 64.

attempts:INT
How many times to we re-transmit a DNS request before giving up on it? Defaults to 3.

randomize-case:INT
If nonzero, we randomize the case on outgoing DNS requests and make sure that replies have the same case as our requests. This so-called "0x20 hack" can help prevent some otherwise simple active events against DNS. Defaults to 1.

bind-to:ADDRESS
If provided, we bind to the given address whenever we send packets to a nameserver. As of Libevent 2.0.4-alpha, it only applied to subsequent nameserver entries.

initial-probe-timeout:FLOAT
When we decide that a nameserver is down, we probe it with exponentially decreasing frequency to see if it has come back up. This option configures the first timeout in the series, in seconds. Defaults to 10.

getaddrinfo-allow-skew:FLOAT
When evdns_getaddrinfo() requests both an IPv4 address and an IPv6 address, it does so in separate DNS request packets, since some servers can't handle both requests in one packet. Once it has an answer for one address type, it waits a little while to see if an answer for the other one comes in. This option configures how long to wait, in seconds. Defaults to 3 seconds.

Unrecognized tokens and options are ignored.

Configuring evdns manually

If you want even more fine-grained control over evdns's behavior, you can use these functions:

Interface

```
int evdns_base_nameserver_sockaddr_add(struct evdns_base *base,
                                       const struct sockaddr *sa, ev_socklen_t len,
                                       unsigned flags);
int evdns_base_nameserver_ip_add(struct evdns_base *base,
                                 const char *ip_as_string);
int evdns_base_load_hosts(struct evdns_base *base, const char *hosts_fname);

void evdns_base_search_clear(struct evdns_base *base);
void evdns_base_search_add(struct evdns_base *base, const char *domain);
void evdns_base_search_ndots_set(struct evdns_base *base, int ndots);

int evdns_base_set_option(struct evdns_base *base, const char *option,
    const char *val);

int evdns_base_count_nameservers(struct evdns_base *base);
```

The evdns_base_nameserver_sockaddr_add() function adds a nameserver to an existing evdns_base by its address. The *flags* argument is currently ignored, and should be 0 for forward-compatibility. The function returns 0 on success and negative on failure. (It was added in Libevent 2.0.7-rc.)

The evdns_base_nameserver_ip_add function adds a nameserver to an existing evdns_base. It takes the nameserver in a text string, either as an IPv4 address, an IPv6 address, an IPv4 address with a port (IPv4:Port), or an IPv6 address with a port ([IPv6]:Port). It returns 0 on success and negative on failure.

The evdns_base_load_hosts() function loads a hosts file (in the same format as /etc/hosts) from hosts_fname. It also returns 0 on success and negative on failure.

The evdns_base_search_clear() function removes all current search suffixes (as configured by the *search* option) from the evdns_base; the evdns_base_search_add() function adds a suffix.

The evdns_base_set_option() function sets a given option to a given value in the evdns_base. Each one is given as a string. (Before Libevent 2.0.3, the option name needed to have a colon after it.)

If you've just parsed a set of configuration files and want to see if any nameservers were added, you can use evdns_base_count_nameserve to see how many there are.

Library-side configuration

There are a couple of functions you can use to specify library-wide settings for the evdns module:

Interface

```
typedef void (*evdns_debug_log_fn_type)(int is_warning, const char *msg);
void evdns_set_log_fn(evdns_debug_log_fn_type fn);
void evdns_set_transaction_id_fn(ev_uint16_t (*fn)(void));
```

For historical reasons, the evdns subsystem does its own logging; you can use evdns_set_log_fn() to give it a callback that does something with its messages besides discard them.

For security, evdns needs a good source of random numbers: it uses this to pick hard-to-guess transaction IDs and to randomize queries when using the 0x20 hack. (See the "randomize-case" option for more info here.) Older versions of Libevent, did not provide a secure RNG of its own, however. You can give evdns a better random number generator by calling evdns_set_transaction_id_fn and giving it a function that returns a hard-to-predict two-byte unsigned integer.

In Libevent 2.0.4-alpha and later, Libevent uses its own built-in secure RNG; evdns_set_transaction_id_fn() has no effect.

Low-level DNS interfaces

Occasionally, you'll want the ability to launch specific DNS requests with more fine-grained control than you get from evdns_getaddrinfo Libevent gives you some interfaces to do that.

Missing features Right now, Libevent's DNS support lacks a few features that you'd expect from a low-level DNS system, like support for arbitrary request types and TCP requests. If you need features that evdns doesn't have, please consider contributing a patch. You might also look into a more full-featured DNS library like c-ares.

Interface

```
#define DNS_QUERY_NO_SEARCH /* ... */

#define DNS_IPv4_A          /* ... */
#define DNS_PTR             /* ... */
#define DNS_IPv6_AAAA       /* ... */

typedef void (*evdns_callback_type)(int result, char type, int count,
    int ttl, void *addresses, void *arg);

struct evdns_request *evdns_base_resolve_ipv4(struct evdns_base *base,
    const char *name, int flags, evdns_callback_type callback, void *ptr);
struct evdns_request *evdns_base_resolve_ipv6(struct evdns_base *base,
    const char *name, int flags, evdns_callback_type callback, void *ptr);
struct evdns_request *evdns_base_resolve_reverse(struct evdns_base *base,
    const struct in_addr *in, int flags, evdns_callback_type callback,
    void *ptr);
struct evdns_request *evdns_base_resolve_reverse_ipv6(
    struct evdns_base *base, const struct in6_addr *in, int flags,
    evdns_callback_type callback, void *ptr);
```

These resolve functions initiate a DNS request for a particular record. Each takes an evdns_base to use for the request, a resource to look up (either a hostname for forward lookups, or an address for reverse lookups), a set of flags to determine how to do the lookup, a callback to invoke when the lookup is done, and a pointer to pass to the user-supplied callback.

The *flags* argument is either 0 or DNS_QUERY_NO_SEARCH to explicitly suppress searching in the list of search if the original search fails. DNS_QUERY_NO_SEARCH has no effect for reverse lookups, since those never do searching.

When the request is done---either successfully or not---the callback function will be invoked. The callback takes a *result* that indicates success or an error code (see DNS Errors table below), a record type (one of DNS_IPv4_A, DNS_IPv6_AAAA, or DNS_PTR), the number of records in *addresses*, a time-to-live in seconds, the addresses themselves, and the user-supplied argument pointer.

The *addresses* argument to the callback is NULL in the event of an error. For a PTR record, it's a NUL-terminated string. For IPv4 records, it is an array of four-byte values in network order. For IPv6 records, it is an array of 16-byte records in network order. (Note that the number of addresses can be 0 even if there was no error. This can happen when the name exists, but it has no records of the requested type.)

The errors codes that can be passed to the callback are as follows:

DNS Errors [options="header",width="70%"

Code	Meaning
DNS_ERR_NONE	No error occurred
DNS_ERR_FORMAT	The server didn't understand the query
DNS_ERR_SERVERFAILED	The server reported an internal error
DNS_ERR_NOTEXIST	There was no record with the given name
DNS_ERR_NOTIMPL	The server doesn't understand this kind of query
DNS_ERR_REFUSED	The server rejected the query for policy reasons
DNS_ERR_TRUNCATED	The DNS record wouldn't fit in a UDP packet
DNS_ERR_UNKNOWN	Unknown internal error
DNS_ERR_TIMEOUT	We waited too long for an answer
DNS_ERR_SHUTDOWN	The user asked us to shut down the evdns system
DNS_ERR_CANCEL	The user asked us to cancel this request
DNS_ERR_NODATA	The response arrived, but contained no answers

(DNS_ERR_NODATA was new in 2.0.15-stable.)

You can decode these error codes to a human-readable string with:

Interface

```
const char *evdns_err_to_string(int err);
```

Each resolve function returns a pointer to an opaque *evdns_request* structure. You can use this to cancel the request at any point before the callback is invoked:

Interface

```
void evdns_cancel_request(struct evdns_base *base,
    struct evdns_request *req);
```

Canceling a request with this function makes its callback get invoked with the DNS_ERR_CANCEL result code.

Suspending DNS client operations and changing nameservers

Sometimes you want to reconfigure or shut down the DNS subsystem without affecting in-flight DNS request too much.

Interface

```
int evdns_base_clear_nameservers_and_suspend(struct evdns_base *base);
int evdns_base_resume(struct evdns_base *base);
```

If you call evdns_base_clear_nameservers_and_suspend() on an evdns_base, all nameservers are removed, and pending requests are left in limbo until later you re-add nameservers and call evdns_base_resume().

These functions return 0 on success and -1 on failure. They were introduced in Libevent 2.0.1-alpha.

DNS server interfaces

Libevent provides simple functionality for acting as a trivial DNS server and responding to UDP DNS requests.

This section assumes some familiarity with the DNS protocol.

Creating and closing a DNS server

Interface

```
struct evdns_server_port *evdns_add_server_port_with_base(
    struct event_base *base,
    evutil_socket_t socket,
    int flags,
    evdns_request_callback_fn_type callback,
    void *user_data);

typedef void (*evdns_request_callback_fn_type)(
    struct evdns_server_request *request,
    void *user_data);

void evdns_close_server_port(struct evdns_server_port *port);
```

To begin listening for DNS requests, call evdns_add_server_port_with_base(). It takes an event_base to use for event handling; a UDP socket to listen on; a flags variable (always 0 for now); a callback function to call when a new DNS query is received; and a pointer to user data that will be passed to the callback. It returns a new evdns_server_port object.

When you are done with the DNS server, you can pass it to evdns_close_server_port().

The evdns_add_server_port_with_base() function was new in 2.0.1-alpha; evdns_close_server_port() was introduced in 1.3.

Examining a DNS request

Unfortunately, Libevent doesn't currently provide a great way to look at DNS requests via a programmatic interface. Instead, you're stuck including event2/dns_struct.h and looking at the evdns_server_request structure manually.

It would be great if a future version of Libevent provided a better way to do this.

Interface

```
struct evdns_server_request {
        int flags;
        int nquestions;
        struct evdns_server_question **questions;
};
#define EVDNS_QTYPE_AXFR 252
#define EVDNS_QTYPE_ALL  255
struct evdns_server_question {
        int type;
        int dns_question_class;
        char name[1];
};
```

The *flags* field of the request contains the DNS flags set in the request; the *nquestions* field is the number of questions in the request; and *questions* is an array of pointers to struct evdns_server_question. Each evdns_server_question includes the resource type of the request (see below for a list of EVDNS_*_TYPE macros), the class of the request (typically EVDNS_CLASS_INET), and the name of the requested hostname.

These structures were introduced in Libevent 1.3. Before Libevent 1.4, dns_question_class was called "class", which made trouble for the C++ people. C programs that still use the old "class" name will stop working in a future release.

Interface

```
int evdns_server_request_get_requesting_addr(struct evdns_server_request *req,
        struct sockaddr *sa, int addr_len);
```

Sometimes you'll want to know which address made a particular DNS request. You can check this by calling evdns_server_request_get_r on it. You should pass in a sockaddr with enough storage to hold the address: struct sockaddr_storage is recommended.

This function was introduced in Libevent 1.3c.

Responding to DNS requests

Every time your DNS server receives a request, the request is passed to the callback function you provided, along with your user_data pointer. The callback function must either respond to the request, ignore the request, or make sure that the request is *eventually* answered or ignored.

Before you respond to a request, you can add one or more answers to your response:

Interface

```
int evdns_server_request_add_a_reply(struct evdns_server_request *req,
    const char *name, int n, const void *addrs, int ttl);
int evdns_server_request_add_aaaa_reply(struct evdns_server_request *req,
    const char *name, int n, const void *addrs, int ttl);
int evdns_server_request_add_cname_reply(struct evdns_server_request *req,
    const char *name, const char *cname, int ttl);
```

The functions above all add a single RR (of type A, AAAA, or CNAME respectively) to the answers section of a DNS reply for the request *req*. In each case the argument *name* is the hostname to add an answer for, and *ttl* is the time-to-live value of the answer in seconds. For A and AAAA records, *n* is the number of addresses to add, and *addrs* is a pointer to the raw addresses, either given as a sequence of n*4 bytes for IPv4 addresses in an A record, or as a sequence of n*16 bytes for IPv6 addresses in an AAAA record.

These functions return 0 on success and -1 on failure.

Interface

```
int evdns_server_request_add_ptr_reply(struct evdns_server_request *req,
    struct in_addr *in, const char *inaddr_name, const char *hostname,
    int ttl);
```

This function adds a PTR record to the answer section of a request. The arguments *req* and *ttl* are as above. You must provide exactly one of *in* (an IPv4 address) or *inaddr_name* (an address in the .arpa domain) to indicate which address you're providing a response for. The *hostname* argument is the answer for the PTR lookup.

Interface

```
#define EVDNS_ANSWER_SECTION 0
#define EVDNS_AUTHORITY_SECTION 1
#define EVDNS_ADDITIONAL_SECTION 2

#define EVDNS_TYPE_A       1
#define EVDNS_TYPE_NS      2
#define EVDNS_TYPE_CNAME   5
#define EVDNS_TYPE_SOA     6
```

```
#define EVDNS_TYPE_PTR    12
#define EVDNS_TYPE_MX     15
#define EVDNS_TYPE_TXT    16
#define EVDNS_TYPE_AAAA   28

#define EVDNS_CLASS_INET   1

int evdns_server_request_add_reply(struct evdns_server_request *req,
    int section, const char *name, int type, int dns_class, int ttl,
    int datalen, int is_name, const char *data);
```

This function adds an arbitrary RR to the DNS reply of a request *req*. The *section* argument describes which section to add it to, and should be one of the EVDNS_*_SECTION values. The *name* argument is the name field of the RR. The *type* argument is the *type* field of the RR, and should be one of the EVDNS_TYPE_* values if possible. The *dns_class* argument is the class field of the RR, and should generally be EVDNS_CLASS_INET. The *ttl* argument is the time-to-live in seconds of the RR. The rdata and rdlength fields of the RR will be generated from the *datalen* bytes provided in *data*. If is_name is true, the data will be encoded as a DNS name (i.e., with DNS name compression). Otherwise, it's included verbatim.

Interface

```
int evdns_server_request_respond(struct evdns_server_request *req, int err);
int evdns_server_request_drop(struct evdns_server_request *req);
```

The evdns_server_request_respond() function sends a DNS response to a request, including all of the RRs that you attached to it, with the error code *err*. If you get a request that you don't want to respond to, you can ignore it by calling evdns_server_request_drop() on it to release all the associated memory and bookkeeping structures.

Interface

```
#define EVDNS_FLAGS_AA  0x400
#define EVDNS_FLAGS_RD  0x080

void evdns_server_request_set_flags(struct evdns_server_request *req,
                                    int flags);
```

If you want to set any flags on your response message, you can call this function at any time before you send the response.

All the functions in this section were introduced in Libevent 1.3, except for evdns_server_request_set_flags() which first appeared in Libevent 2.0.1-alpha.

DNS Server example

Example: A trivial DNS responder

```
#include <event2/dns.h>
#include <event2/dns_struct.h>
#include <event2/util.h>
#include <event2/event.h>

#include <sys/socket.h>

#include <stdio.h>
#include <string.h>
#include <assert.h>

/* Let's try binding to 5353.  Port 53 is more traditional, but on most
   operating systems it requires root privileges. */
#define LISTEN_PORT 5353

#define LOCALHOST_IPV4_ARPA "1.0.0.127.in-addr.arpa"
#define LOCALHOST_IPV6_ARPA ("1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0."         \
```

```
                              "0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa")

const ev_uint8_t LOCALHOST_IPV4[] = { 127, 0, 0, 1 };
const ev_uint8_t LOCALHOST_IPV6[] = { 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,1 };

#define TTL 4242

/* This toy DNS server callback answers requests for localhost (mapping it to
   127.0.0.1 or ::1) and for 127.0.0.1 or ::1 (mapping them to localhost).
 */
void server_callback(struct evdns_server_request *request, void *data)
{
    int i;
    int error=DNS_ERR_NONE;
    /* We should try to answer all the questions.  Some DNS servers don't do
       this reliably, though, so you should think hard before putting two
       questions in one request yourself. */
    for (i=0; i < request->nquestions; ++i) {
        const struct evdns_server_question *q = request->questions[i];
        int ok=-1;
        /* We don't use regular strcasecmp here, since we want a locale-
           independent comparison. */
        if (0 == evutil_ascii_strcasecmp(q->name, "localhost")) {
            if (q->type == EVDNS_TYPE_A)
                ok = evdns_server_request_add_a_reply(
                       request, q->name, 1, LOCALHOST_IPV4, TTL);
            else if (q->type == EVDNS_TYPE_AAAA)
                ok = evdns_server_request_add_aaaa_reply(
                       request, q->name, 1, LOCALHOST_IPV6, TTL);
        } else if (0 == evutil_ascii_strcasecmp(q->name, LOCALHOST_IPV4_ARPA)) {
            if (q->type == EVDNS_TYPE_PTR)
                ok = evdns_server_request_add_ptr_reply(
                       request, NULL, q->name, "LOCALHOST", TTL);
        } else if (0 == evutil_ascii_strcasecmp(q->name, LOCALHOST_IPV6_ARPA)) {
            if (q->type == EVDNS_TYPE_PTR)
                ok = evdns_server_request_add_ptr_reply(
                       request, NULL, q->name, "LOCALHOST", TTL);
        } else {
            error = DNS_ERR_NOTEXIST;
        }
        if (ok<0 && error==DNS_ERR_NONE)
            error = DNS_ERR_SERVERFAILED;
    }
    /* Now send the reply. */
    evdns_server_request_respond(request, error);
}

int main(int argc, char **argv)
{
    struct event_base *base;
    struct evdns_server_port *server;
    evutil_socket_t server_fd;
    struct sockaddr_in listenaddr;

    base = event_base_new();
    if (!base)
        return 1;

    server_fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (server_fd < 0)
        return 2;
    memset(&listenaddr, 0, sizeof(listenaddr));
```

```
    listenaddr.sin_family = AF_INET;
    listenaddr.sin_port = htons(LISTEN_PORT);
    listenaddr.sin_addr.s_addr = INADDR_ANY;
    if (bind(server_fd, (struct sockaddr*)&listenaddr, sizeof(listenaddr))<0)
        return 3;
    /*The server will hijack the event loop after receiving the first request if the socket ↩
     is blocking*/
    if(evutil_make_socket_nonblocking(server_fd)<0)
        return 4;
    server = evdns_add_server_port_with_base(base, server_fd, 0,
                                             server_callback, NULL);

    event_base_dispatch(base);

    evdns_close_server_port(server);
    event_base_free(base);

    return 0;
}
```

Obsolete DNS interfaces

Obsolete Interfaces

```
void evdns_base_search_ndots_set(struct evdns_base *base,
                                 const int ndots);
int evdns_base_nameserver_add(struct evdns_base *base,
    unsigned long int address);
void evdns_set_random_bytes_fn(void (*fn)(char *, size_t));

struct evdns_server_port *evdns_add_server_port(evutil_socket_t socket,
    int flags, evdns_request_callback_fn_type callback, void *user_data);
```

Calling evdns_base_search_ndots_set() is equivalent to using evdns_base_set_option() with the "ndots" option.

The evdns_base_nameserver_add() function behaves as evdns_base_nameserver_ip_add(), except it can only add nameservers with IPv4 addresses. It takes them, idiosyncratically, as four bytes in network order.

Before Libevent 2.0.1-alpha, there was no way to specify a event base for a DNS server port. You had to use evdns_add_server_port() instead, which took the default event_base.

From Libevent 2.0.1-alpha through 2.0.3-alpha, you could use evdns_set_random_bytes_fn to specify a function to use for generating random numbers instead of evdns_set_transaction_id_fn. It no longer has any effect, now that Libevent provides its own secure RNG.

The DNS_QUERY_NO_SEARCH flag has also been called DNS_NO_SEARCH.

Before Libevent 2.0.1-alpha, there was no separate notion of an evdns_base: all information in the evdns subsystem was stored globally, and the functions that manipulated it took no evdns_base as an argument. They are all now deprecated, and declared only in event2/dns_compat.h. They are implemented via a single global evdns_base; you can access this base by calling the evdns_get_global_base() function introduced in Libevent 2.0.3-alpha.

Current function	Obsolete global-evdns_base version
event_base_new()	evdns_init()
evdns_base_free()	evdns_shutdown()
evdns_base_nameserver_add()	evdns_nameserver_add()
evdns_base_count_nameservers()	evdns_count_nameservers()
evdns_base_clear_nameservers_and_suspend()	evdns_clear_nameservers_and_suspend()
evdns_base_resume()	evdns_resume()

Current function	Obsolete global-evdns_base version
evdns_base_nameserver_ip_add()	evdns_nameserver_ip_add()
evdns_base_resolve_ipv4()	evdns_resolve_ipv4()
evdns_base_resolve_ipv6()	evdns_resolve_ipv6()
evdns_base_resolve_reverse()	evdns_resolve_reverse()
evdns_base_resolve_reverse_ipv6()	evdns_resolve_reverse_ipv6()
evdns_base_set_option()	evdns_set_option()
evdns_base_resolv_conf_parse()	evdns_resolv_conf_parse()
evdns_base_search_clear()	evdns_search_clear()
evdns_base_search_add()	evdns_search_add()
evdns_base_search_ndots_set()	evdns_search_ndots_set()
evdns_base_config_windows_nameservers()	evdns_config_windows_nameservers()

The EVDNS_CONFIG_WINDOWS_NAMESERVERS_IMPLEMENTED macro is defined if and only if evdns_config_windows_name
is available.