```
Doc. no:  N2054=06-0124
Date:     2006-09-08
Project:  Programming Language C++
```


```
Reply to: Christopher Kohlhoff <chris@kohlhoff.com>
```


Networking Library Proposal for TR2

Table of Contents

1. Overview

This document proposes a networking library for TR2. The library supports several levels of use, ranging from entry-level to advanced.

To give some idea of the flavour of the proposed library, consider the following sample code. This is part of a server program that echoes characters it receives back to the client in upper case:

```
namespace sys = std::tr2::sys;

template <typename Iterator>
void uppercase(Iterator begin, Iterator end)
{
  std::locale loc("");
  for (Iterator iter = begin; iter != end; ++iter)
    *iter = std::toupper(*iter, loc);
}

void do_sync(
    sys::ip::tcp::socket& socket,
    std::vector<char>& buffer_space)
{
  try
  {
    for (;;)
    {
      std::size_t count = socket.read_some(sys::buffer(buffer_space));
      uppercase(buffer_space.begin(), buffer_space.begin() + count);
      sys::write(socket, sys::buffer(buffer_space, count));
    }
  }
  catch (sys::system_error& e)
  {
  }
}
```

The synchronous approach used above is straightforward to understand and easy for programmers at any level of ability to write.

Next, the equivalent code developed using asynchronous operations:

```
void do_async(
    sys::ip::tcp::socket& socket,
    std::vector<char>& buffer_space)
{
  socket.async_read_some(sys::buffer(buffer_space),
      std::tr1::bind(handle_read, _1, _2,
        std::tr1::ref(socket), std::tr1::ref(buffer_space)));
}

void handle_read(
    sys::error_code ec,
    std::size_t count,
    sys::ip::tcp::socket& socket,
    std::vector<char>& buffer_space)
{
  if (!ec)
  {
    uppercase(buffer_space.begin(), buffer_space.begin() + count);
    sys::async_write(socket, sys::buffer(buffer_space, count),
        std::tr1::bind(handle_write, _1,
          std::tr1::ref(socket), std::tr1::ref(buffer_space)));
  }
}

void handle_write(
    sys::error_code ec,
    sys::ip::tcp::socket& socket,
    std::vector<char>& buffer_space)
{
  if (!ec)
  {
    socket.async_read_some(sys::buffer(buffer_space),
        std::tr1::bind(handle_read, _1, _2,
          std::tr1::ref(socket), std::tr1::ref(buffer_space)));
  }
```

```
}
```

This code may appear more complex due to the inverted flow of control, but it allows a knowledgeable programmer to write code that will scale to a great many concurrent connections. The synchronous code requires one thread for each connection, and on most platforms threads are a limited resource. The asynchronous approach described in this proposal has been exercised in production HTTP servers to thousands of concurrent connections, and similar echo servers have been tested to tens of thousands, while using only one thread.

2. Motivation and Scope

2.1. Scope

Problem areas addressed by this proposal include:

- Networking using TCP and UDP, including support for multicast.
- Client and server applications.
- Scalability to handle many concurrent connections.
- Protocol independence between IPv4 and IPv6.
- Name resolution (i.e. DNS).
- Timers.

Features that are considered outside the scope of this proposal include:

- Protocol implementations such as HTTP, SMTP or FTP.
- Encryption (e.g. SSL, TLS).
- Operating system specific demultiplexing APIs.
- Support for realtime environments.
- QoS-enabled sockets.
- Other TCP/IP protocols such as ICMP.
- Functions and classes for enumerating network interfaces.

2.2. Target Audience

The bulk of the library interface is intended for use by developers with at least some understanding of networking concepts (or a willingness to learn). A high level iostreams interface supports simple use cases and permits novices to develop network code without needing to get into too much depth.

2.3. Reference Implementation

The Boost.Asio library, from which this proposal is derived, has been deployed in a number of production systems, such as internet-facing HTTP servers, instant messaging gateways and finance applications.

The Boost.Asio library has been used on the following platforms:

- Win32 using Visual C++ 7.1 and Visual C++ 8.0.
- Win32 using Borland C++Builder 6 patch 4.
- Win32 using MinGW.
- Win32 using Cygwin.
- Linux (2.4 or 2.6 kernels) using g++ 3.3 or later.
- Solaris using g++ 3.3 or later.
- Mac OS X 10.4 using g++ 3.3 or later.
- QNX Neutrino 6.3 using g++ 3.3 or later.
- FreeBSD using g++ 3.3 or later.

Boost.Asio may be obtained from http://asio.sourceforge.net.

2.4. Related Work

The interface is based on the BSD sockets API, which is widely implemented and supported by extensive literature. It is also used as the basis of networking APIs in other languages (e.g. Java). Unsafe practices of the BSD sockets API, e.g. lack of compile-time type safety, are not included.

Asynchronous support is derived from the Proactor design pattern as implemented by the ADAPTIVE Communication Environment [ACE], and is influenced by the design of the Symbian C++ sockets API [SYMBIAN], which supports synchronous and asynchronous operations side-by-side. The Microsoft .NET socket classes [MS-NET] and the Extended Sockets API [ES-API] developed by The Open Group support similar styles of network programming.

3. Impact On the Standard

This is a pure library proposal. It does not add any new language features, nor does it alter any existing standard library headers.

This library can be implemented using compilers that conform to the C++03 standard. An implementation of this library requires operating system-specific functions that lie outside the C++03 standard.

3.1. Relationship to TR1

This proposal uses the TR1 libraries for fixed size arrays and regular expressions. Programs developed using the proposed library typically make extensive use of TR1 function object binders.

3.2. Relationship to Threading and Memory Model Proposals

This proposal does not require, and would not be coupled to, hypothetical standard library support for threading. The interface is intended to support implementations on platforms where threads are not available.

However, the library interface is designed to allow the effective utilisation of threading if available, and its behaviour with respect to threads is clearly defined. In particular, the proposal will attempt to address:

- Thread safety of classes and functions defined in the interface.
- The threads from which an implementation is permitted to call user code, and when.
- The relationship between asynchronous operation initiation, completion, and inter-thread memory visibility.

3.3. Relationship to Date-Time Library Proposal

This proposal uses classes defined in the Proposal to Add Date-Time to the C++ Standard Library [N1900].

3.4. Relationship to Filesystem Library Proposal

The classes defined in the Diagnostics Library chapter of the Filesystem Library Proposal [N1975], and updated by [N2066], are used in this proposal.

4. Design Decisions

4.1. BSD Sockets

The proposal includes a low-level interface based on the BSD sockets API, which is widely implemented and supported by extensive literature. It is also used as the basis for networking APIs in other languages, like Java.

This low-level interface is designed to support the development of efficient and scalable applications. For example, it permits programmers to exert finer control over the number of system calls, avoid redundant data copying, minimise the use of resources like threads, and so on.

Unsafe and error prone aspects of the BSD sockets API not included. For example, type safety is enforced at compile-time by using classes to represent sockets rather than `int`.

4.2. Protocol Independence

The library described in this proposal supports TCP and UDP for both IP versions 4 and 6. It enables the development of protocol independent applications. That is, the decision of whether to use IPv4 or IPv6 can be deferred until runtime.

4.3. Asynchronous Operations

The proposed library offers side-by-side support for synchronous and asynchronous operations. The asynchronous support is based on the Proactor design pattern [POSA2], and the advantages and disadvantages of this approach, when compared to a synchronous-only or Reactor approach, are outlined below.

4.3.1. Advantages

— Portability.

Many operating systems offer a native asynchronous I/O API (such as overlapped I/O on *Windows*) as the preferred option for developing high performance network applications. The proposed library may be implemented in terms of native asynchronous I/O. However, if native support is not available, the library may also be implemented using synchronous event demultiplexors that typify the Reactor pattern, such as *POSIX* `select()`.

— Decoupling threading from concurrency.

Long-duration operations are performed asynchronously by the implementation on behalf of the application. Consequently applications do not need to spawn many threads in order to increase concurrency.

— Performance and scalability.

Implementation strategies such as thread-per-connection (which a synchronous-only approach would require) can degrade system performance, due to increased context switching, synchronisation and data movement among CPUs. With asynchronous operations it is possible to avoid the cost of context switching by minimising the number of operating system threads — typically a limited resource — and only activating the logical threads of control that have events to process.

— Simplified application synchronisation.

Asynchronous operation completion handlers can be written as though they exist in a single-threaded environment, and so application logic can be developed with little or no concern for synchronisation issues.

— Function composition.

Function composition refers to the implementation of functions to provide a higher-level operation, such as sending a message in a particular format. Each function is implemented in terms of multiple calls to lower-level read or write operations.

For example, consider a protocol where each message consists of a fixed-length header followed by a variable length body, where the length of the body is specified in the header. A hypothetical read_message operation could be implemented using two lower-level reads, the first to receive the header and, once the length is known, the second to receive the body.

To compose functions in an asynchronous model, asynchronous operations can be chained together. That is, a completion handler for one operation can initiate the next. Starting the first call in the chain can be encapsulated so that the caller need not be aware that the higher-level operation is implemented as a chain of asynchronous operations.

The ability to compose new operations in this way simplifies the development of higher levels of abstraction above a networking library, such as functions to support a specific protocol.

4.3.2. Disadvantages

— Program complexity.

It is more difficult to develop applications using asynchronous mechanisms due to the separation in time and space between operation initiation and completion. Applications may also be harder to debug due to the inverted flow of control.

— Memory usage.

Buffer space must be committed for the duration of a read or write operation, which may continue indefinitely, and a separate buffer is required for each concurrent operation. The Reactor pattern, on the other hand, does not require buffer space until a socket is ready for reading or writing.

4.4. IOStreams

The proposal includes classes that implement iostreams on top of sockets. These hide away the complexities associated with endpoint resolution, protocol independence, etc. For example, to create a connection one might simply write:

```
ip::tcp::iostream stream("www.boost.org", "http");
if (!stream)
{
```

```
  // Can't connect.
}
```

The iostream class can also be used in conjunction with an acceptor to create simple servers. For example:

```
io_service ios;
ip::tcp::endpoint endpoint(tcp::v4(), 80);
ip::tcp::acceptor(ios, endpoint);
for (;;)
{
  ip::tcp::iostream stream;
  acceptor.accept(*stream.rdbuf());
  ...
}
```

Note: these iostream templates only support `char`, not `wchar_t`, and do not perform any code conversion.

4.5. POSIX, Windows and Extensibility

This proposal defines two levels of conformance:

- Basic interfaces which all implementations must follow.
- Additional member functions that provide extensibility. It is intended that *POSIX* and *Windows* implementations will provide these.

Implementations on platforms that have a sockets API similar to *POSIX* and *Windows* are also encouraged to provide the additional member functions.

The rationale is to allow the implementation to be portably extended by a program (or implementor) to add additional:

- protocols, such as UNIX domain sockets, infrared or Bluetooth;
- socket options; and
- I/O control commands.

Programs that do not require this extensibility should be portable to all platforms that implement the library.

4.6. Threads

4.6.1. Thread Safety

In general, it is safe to make concurrent use of distinct objects, but unsafe to make concurrent use of a single object. However, types such as `io_service` provide a stronger guarantee that it is safe to use a single object concurrently.

4.6.2. Internal Threads

The implementation of this library for a particular platform may make use of one or more internal threads to emulate asynchronicity. As far as possible, these threads must be invisible to the library user. In particular, the threads:

- must not call the user's code directly; and
- must block all signals.

This approach is complemented by the following guarantee:

- Asynchronous completion handlers will only be called from threads that are currently calling io_service::run().

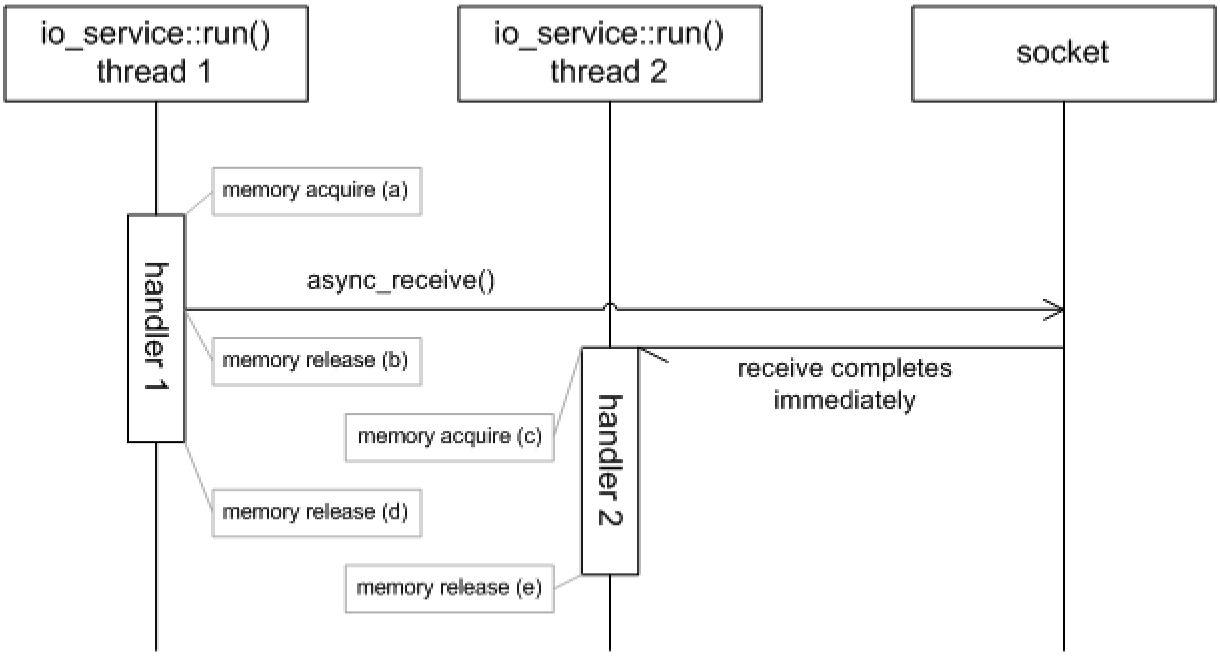
Consequently, it is the library user's responsibility to create and manage all threads to which the notifications will be delivered.

The reasons for this approach include:

- By only calling io_service::run() from a single thread, the user's code can avoid the development complexity associated with synchronisation. For example, a library user can implement scalable servers that are single-threaded (from the user's point of view).
- A library user may need to perform initialisation in a thread shortly after the thread starts and before any other application code is executed. For example, users of Microsoft's COM must call CoInitializeEx before any other COM operations can be called from that thread.
- The library interface is decoupled from interfaces for thread creation and management, and permits implementations on platforms where threads are not available.

4.6.3. Memory Visibility

The following diagram outlines one possible strategy for managing memory visibility. It is intended that programs that make use of implicit or explicit strands should not have to directly include memory barriers or synchronisation mechanisms.



4.7. Strands

A strand is defined as a strictly sequential invocation of event handlers (i.e. no concurrent invocation). Use of strands allows execution of code in a multithreaded program without the need for explicit locking (e.g. using mutexes).

Strands may be either implicit or explicit, as illustrated by the following alternative approaches:

- Calling io_service::run() from only one thread means all event handlers execute in an implicit strand, due to the io_service's guarantee that handlers are only invoked from inside run().
- Where there is a single chain of asynchronous operations associated with a connection (e.g. in a half duplex protocol implementation like HTTP) there is no possibility of concurrent execution of the handlers. This is an implicit strand.
- An explicit strand is an instance of `io_service::strand`. All event handler function objects need to be wrapped using `io_service::strand::wrap()` or otherwise posted/dispatched through the `io_service::strand` object.

In the case of composed asynchronous operations, such as `async_read()` or `async_read_until()`, if a completion handler goes through a strand, then all intermediate handlers should also go through the same strand. This is needed to ensure thread safe access for any objects that are shared between the caller and the composed operation (in the case of `async_read()` it's the socket, which the caller can close() to cancel the operation). This is done by having hook functions for all intermediate handlers which forward the calls to the customisable hook associated with the final handler:

```
struct my_handler
{
  void operator()() { ... }
};

template<class F>
void io_handler_invoke(F f, my_handler*)
{
  // Do custom invocation here.
  // Default implementation calls f();
}
```

The `io_service::strand::wrap()` function creates a new completion handler that defines `io_handler_invoke` so that the function object is executed through the strand.

4.8. Buffers

To allow the development of efficient network applications, this proposal includes support for scatter-gather operations. These operations involve one or more buffers (where each buffer is a contiguous region of memory):

- A scatter-read receives data into multiple buffers.
- A gather-write transmits multiple buffers.

Therefore we require an abstraction to represent a collection of buffers. The approach used in this proposal is to define a type (actually two types) to represent a single buffer. These can be stored in a container, which may be passed to the scatter-gather operations.

A buffer, as a contiguous region of memory, can be represented by an address and size in bytes. There is a distinction between modifiable memory (called mutable in this proposal) and non-modifiable memory (where the latter is created from the storage for a const-qualified variable). These two types could therefore be defined as follows:

```
typedef std::pair<void*, std::size_t> mutable_buffer;
typedef std::pair<const void*, std::size_t> const_buffer;
```

Here, a mutable_buffer would be convertible to a const_buffer, but conversion in the opposite direction is not valid.

However, this proposal library does not use the above definitions, but instead defines two classes: `mutable_buffer` and `const_buffer`. The goal of these is to provide an opaque representation of contiguous memory, where:

- Types behave as std::pair would in conversions. That is, a `mutable_buffer` is convertible to a `const_buffer`, but the opposite conversion is disallowed.
- There is protection against buffer overruns. Given a buffer instance, a user can only create another buffer representing the same range of memory or a sub-range of it. To provide further safety, the library also includes mechanisms for automatically determining the size of a buffer from an array, `tr1::array` or `std::vector` of POD elements, or from a `std::string`.
- Type safety violations must be explicitly requested using the `buffer_cast` function. In general an application should never need to do this, but it is required by the library implementation to pass the raw memory to the underlying operating system functions.

Finally, multiple buffers can be passed to scatter-gather operations (such as `std::tr2::sys::read` or `std::tr2::sys::write`) by putting the buffer objects into a container. The `MutableBufferSequence` and `ConstBufferSequence` concepts have been defined so that containers such as `std::vector`, `std::list`, `std::vector` or `tr1::array` can be used.

4.9. Custom Memory Allocation

Many asynchronous operations need to allocate an object to store state associated with the operation. For example, a Win32 implementation needs `OVERLAPPED`-derived objects to pass to Win32 API functions.

Furthermore, programs typically contain easily identifiable chains of asynchronous operations. A half duplex protocol implementation (e.g. an HTTP server) would have a single chain of operations per client (receives followed by sends). A full duplex protocol implementation would have two chains executing in parallel. Programs should be able to leverage this knowledge to reuse memory for all asynchronous operations in a chain.

Given a copy of a user-defined `Handler` object `h`, if the implementation needs to allocate memory associated with that handler it will execute the code:

```
void* pointer = io_handler_allocate(size, &h);
```

Similarly, to deallocate the memory it will execute:

```
io_handler_deallocate(pointer, size, &h);
```

These functions are located using argument-dependent lookup. The implementation provides default implementations of the above functions in the `std::tr2::sys` namespace:

```
void* io_handler_allocate(size_t, ...);
void io_handler_deallocate(void*, size_t, ...);
```

which are implemented in terms of `::operator new()` and `::operator delete()` respectively.

The implementation guarantees that the deallocation will occur before the associated handler is invoked, which means the memory is ready to be reused for any new asynchronous operations started by the handler.

The custom memory allocation functions may be called from any user-created thread that is calling a library function. The implementation guarantees that, for the asynchronous operations included the library, the implementation will not make concurrent calls to the memory allocation functions for that handler. The implementation will insert appropriate memory barriers to ensure correct memory visibility should allocation functions need to be called from different threads.

4.10. Line-Based Protocols

Many commonly-used internet protocols are line-based, which means that they have protocol elements that are delimited by the character sequence `"\r\n"`. Examples include HTTP [RFC2616], SMTP [RFC821] and FTP [RFC959].

To more easily permit the implementation of line-based protocols, as well as other protocols that use delimiters, this proposal includes the functions `read_until()` and `async_read_until()`.

The following example illustrates the use of `async_read_until()` in an HTTP server, to receive the first line of an HTTP request from a client:

```
class http_connection
{
  ...

  void start()
  {
    sys::async_read_until(socket_, data_, "\r\n",
        std::tr1::bind(&http_connection::handle_request_line, this, _1));
  }

  void handle_request_line(sys::error_code ec)
  {
    if (!ec)
    {
      std::string method, uri, version;
      char sp1, sp2, cr, lf;
      std::istream is(&data_);
      is.unsetf(std::ios_base::skipws);
      is >> method >> sp1 >> uri >> sp2 >> version >> cr >> lf;
      ...
    }
  }

  ...

  sys::ip::tcp::socket socket_;
  sys::fifobuf data_;
};
```

4.11. Why EOF is an error

- The end of a stream can cause `read`, `async_read`, `read_until` or `async_read_until` functions to violate their contract. E.g. a read of N bytes may finish early due to EOF.
- An EOF error may be used to distinguish the end of a stream from a successful read of size 0.

4.12. Other Design Decisions

The following list summarises some other design decisions made in the development of the library. Some further decisions are also included as commentary in the proposed text.

— The implementation is separated into two layers: "services", and the object-oriented wrappers around them.

— The `read`, `async_read`, `write` and `async_write` functions are included as a solution to the problem of partial reads and writes.

— Signals and system call interruption are hidden in most cases.

— No string constructors for IP address classes.

— Monotonic timers are not included in the proposal, since they are difficult to implement portably, and *POSIX* support for monotonic time is optional. A program can still add support for monotonic timers by using a custom `TimeTraits` implementation.

— EMSGSIZE should not be used to indicate datagram truncation.

Contact the author for more information.

5. Proposed Text for the Standard

Grey-shaded italic text is commentary on the proposal. It is not to be added to the TR.

This clause describes components that C++ programs may use to perform network operations.

The Filesystem Library Proposal [N1975] adds introductory text to the TR to establish a relationship to POSIX [POSIX]. This proposal uses the relationship to similarly define behaviour "as if" implemented by POSIX, except that the requirements for error reporting are stricter than those outlined in [N1975].

The following subclauses describe components for I/O services, timers, buffer management, sockets, endpoint resolution, iostreams, and internet protocol, as summarised in the table below:

Table--Networking library summary

Subclause	Header(s)
Basic I/O services	`<io_service>`
Timers	`<timer>`
Buffers	`<buffer>`
Sockets Socket streams Internet protocol	`<network>`

Throughout this clause, the names of the template parameters are used to express type requirements, as listed in the table below.

Table--Template parameters and type requirements

template parameter name	type requirements
`AcceptHandler`	accept handler
`AsyncReadStream`	buffer-oriented asynchronous read stream
`AsyncWriteStream`	buffer-oriented asynchronous write stream
`CompletionHandler`	completion handler
`ConnectHandler`	connect handler
`ConstBufferSequence`	constant buffer sequence
`ConvertibleToConstBuffer`	convertible to a constant buffer
`ConvertibleToMutableBuffer`	convertible to a mutable buffer
`DatagramSocketService`	datagram socket service
`GettableSocketOption`	gettable socket option
`Handler`	handler
`InternetProtocol`	Internet protocol
`IoControlCommand`	I/O control command
`IoObjectService`	I/O object service
`MutableBufferSequence`	mutable buffer sequence
`Protocol`	protocol
`ReadHandler`	read handler
`ResolveHandler`	resolve handler
`ResolverService`	resolver service
`Service`	service
`SettableSocketOption`	settable socket option
`SocketAcceptorService`	socket acceptor service
`SocketService`	socket service
`StreamSocketService`	stream socket service
`SyncReadStream`	buffer-oriented synchronous read stream
`SyncWriteStream`	buffer-oriented synchronous write stream
`TimerService`	timer service

template parameter name	type requirements
TimeTraits	time traits
WaitHandler	wait handler
WriteHandler	write handler

5.1. Definitions

5.1.1. host byte order

See Host and Network Byte Orders.

5.1.2. network byte order

See Host and Network Byte Orders.

5.1.3. synchronous operation

A synchronous operation is logically executed in the context of the initiating thread. Control is not returned to the initiating thread until the operation completes.

5.1.4. asynchronous operation

An asynchronous operation is logically executed in parallel to the context of the initiating thread. Control is returned immediately to the initiating thread without waiting for the operation to complete. Multiple asynchronous operations may be executed in parallel by a single initiating thread.

5.2. Diagnostics library

The following text is intended as an addition to the Diagnostics Library chapter defined in [N1975] and modified by [N2066].

5.2.1. Header `<error_code>` additions

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace error {

        // errno errors:
        extern const error_code access_denied;
        extern const error_code address_family_not_supported;
        extern const error_code address_in_use;
        extern const error_code address_not_available;
        extern const error_code already_connected;
        extern const error_code already_started;
        extern const error_code broken_pipe;
        extern const error_code bad_descriptor;
        extern const error_code connection_aborted;
        extern const error_code connection_refused;
        extern const error_code connection_reset;
        extern const error_code destination_address_required;
        extern const error_code host_unreachable;
        extern const error_code interrupted;
        extern const error_code invalid_argument;
        extern const error_code invalid_protocol_type;
        extern const error_code message_size;
        extern const error_code network_down;
        extern const error_code network_reset;
        extern const error_code network_unreachable;
        extern const error_code no_buffer_space;
        extern const error_code no_descriptors;
        extern const error_code no_protocol_option;
        extern const error_code no_system_descriptors;
        extern const error_code not_connected;
        extern const error_code not_socket;
        extern const error_code operation_aborted;
        extern const error_code operation_not_supported;
        extern const error_code protocol_not_supported;
```

```
        extern const error_code timed_out;

        // getaddrinfo errors:
        extern const error_code host_not_found;
        extern const error_code host_not_found_try_again;
        extern const error_code service_not_found;

        // miscellaneous errors:
        extern const error_code already_open;
        extern const error_code eof;
        extern const error_code not_found;

      } // namespace error
    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.2.2. errno errors

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace error {

        extern const error_code access_denied;
        extern const error_code address_family_not_supported;
        extern const error_code address_in_use;
        extern const error_code address_not_available;
        extern const error_code already_connected;
        extern const error_code already_started;
        extern const error_code broken_pipe;
        extern const error_code bad_descriptor;
        extern const error_code connection_aborted;
        extern const error_code connection_refused;
        extern const error_code connection_reset;
        extern const error_code destination_address_required;
        extern const error_code host_unreachable;
        extern const error_code interrupted;
        extern const error_code invalid_argument;
        extern const error_code invalid_protocol_type;
        extern const error_code message_size;
        extern const error_code network_down;
        extern const error_code network_reset;
        extern const error_code network_unreachable;
        extern const error_code no_buffer_space;
        extern const error_code no_descriptors;
        extern const error_code no_protocol_option;
        extern const error_code no_system_descriptors;
        extern const error_code not_connected;
        extern const error_code not_socket;
        extern const error_code operation_aborted;
        extern const error_code operation_not_supported;
        extern const error_code protocol_not_supported;
        extern const error_code timed_out;

      } // namespace error
    } // namespace sys
  } // namespace tr2
} // namespace std
```

The meaning of each `error_code` value declared above shall correspond to the *POSIX* equivalent, as defined in the table below. The method of initialisation of each value is implementation-defined.

Table--errno errors

name	*POSIX* equivalent
access_denied	EACCES
address_family_not_supported	EAFNOSUPPORT
address_in_use	EADDRINUSE
address_not_available	EADDRNOTAVAIL
already_connected	EISCONN
already_started	EALREADY

name	*POSIX* equivalent
broken_pipe	EPIPE
bad_descriptor	EBADF
connection_aborted	ECONNABORTED
connection_refused	ECONNREFUSED
connection_reset	ECONNRESET
destination_address_required	EDESTADDRREQ
host_unreachable	EHOSTUNREACH
interrupted	EINTR
invalid_argument	EINVAL
invalid_protocol_type	EPROTOTYPE
message_size	EMSGSIZE
network_down	ENETDOWN
network_reset	ENETRESET
network_unreachable	ENETUNREACH
no_buffer_space	ENOBUFS
no_descriptors	EMFILE
no_protocol_option	ENOPROTOOPT
no_system_descriptors	ENFILE
not_connected	ENOTCONN
not_socket	ENOTSOCK
operation_aborted	ECANCELED
operation_not_supported	EOPNOTSUPP
protocol_not_supported	EPROTONOSUPPORT
timed_out	ETIMEDOUT

[*Note:* The equivalent constants used above are defined in the *POSIX* header file `errno.h`. —*end note*]

5.2.3. `getaddrinfo` errors

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace error {

        extern const error_code host_not_found;
        extern const error_code host_not_found_try_again;
        extern const error_code service_not_found;

      } // namespace error
    } // namespace sys
  } // namespace tr2
} // namespace std
```

The meaning of each `error_code` value declared above shall correspond to the *POSIX* equivalent returned by `getaddrinfo()`, as defined in the table below. The method of initialisation of each value is implementation-defined.

Table--getaddrinfo errors

name	*POSIX* equivalent
host_not_found	EAI_NONAME
host_not_found_try_again	EAI_AGAIN
service_not_found	EAI_SERVICE

[*Note:* The constants `EAI_NONAME`, `EAI_AGAIN` and `EAI_SERVICE` are defined in the *POSIX* header file `netdb.h`. —*end note*]

5.2.4. Miscellaneous errors

```
namespace std {
  namespace tr2 {
```

```
    namespace sys {
      namespace error {

        extern const error_code already_open;
        extern const error_code eof;
        extern const error_code not_found;

      } // namespace error
    } // namespace sys
  } // namespace tr2
} // namespace std
```

The method of initialisation of each `error_code` value declared above is implementation-defined.

5.3. Basic I/O services

5.3.1. Header `<io_service>` synopsis

```
namespace std {
  namespace tr2 {
    namespace sys {

      class io_service;

      class invalid_service_owner;
      class service_already_exists;

      template<class Service> Service& use_service(io_service&);
      template<class Service> void add_service(io_service&, Service*);
      template<class Service> bool has_service(io_service&);

      // default handler hook functions:
      void* io_handler_allocate(size_t s, ...);
      void io_handler_deallocate(void* p, size_t s, ...);
      template<class F> void io_handler_invoke(F f, ...);

      template<class IoObjectService>
        class basic_io_object;

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.3.2. Requirements

5.3.2.1. Handlers

A handler must meet the requirements of `CopyConstructible` types (C++ Std, 20.1.3).

In the table below, `X` denotes a handler class, `h` denotes a value of `X`, `p` denotes a pointer to a block of allocated memory of type `void*`, `s` denotes the size for a block of allocated memory, and `f` denotes a function object taking no arguments.

Table--Handler requirements

expression	return type	assertion/note pre/post-conditions
`using namespace std::tr2::sys;` `io_handler_allocate(s, &h);`	`void*`	Returns a pointer to a block of memory of size `s`. The pointer must satisfy the same alignment requirements as a pointer returned by `::operator new()`. Throws `bad_alloc` on failure. The `io_handler_allocate()` function is located using argument-dependent lookup. The function `std::tr2::sys::io_handler_allocate()` serves as a default if no user-supplied function is available.
`using namespace std::tr2::sys;` `io_handler_deallocate(p, s, &h);`		Frees a block of memory associated with a pointer `p`, of at least size `s`, that was previously allocated using `io_handler_allocate()`. The `io_handler_deallocate()` function is located using

expression	return type	assertion/note pre/post-conditions
		argument-dependent lookup. The function `std::tr2::sys::io_handler_deallocate()` serves as a default if no user-supplied function is available.
`using namespace std::tr2::sys;` `io_handler_invoke(f, &h);`		Causes the function object `f` to be executed as if by calling `f()`. The `io_handler_invoke()` function is located using argument-dependent lookup. The function `std::tr2::sys::io_handler_invoke()` serves as a default if no user-supplied function is available.

5.3.2.2. Completion handler requirements

A completion handler must meet the requirements for a handler. A value `h` of a completion handler class should work correctly in the expression `h()`.

5.3.2.3. Service requirements

A class is a service if it is publicly derived from another service, or if it is a class derived from `io_service::service` and contains a publicly-accessible declaration as follows:

```
static io_service::id id;
```

All services define a one-argument constructor that takes a reference to the `io_service` object that owns the service. This constructor is *explicit*, preventing its participation in automatic conversions. For example:

```
class my_service : public io_service::service
{
public:
  static io_service::id id;
  explicit my_service(io_service& ios);
private:
  virtual void shutdown_service();
  ...
};
```

A service's `shutdown_service` member function must cause all copies of user-defined handler objects that are held by the service to be destroyed.

5.3.2.4. I/O object service requirements

An I/O object service must meet the requirements for a service, as well as the requirements listed below.

In the table below, `X` denotes an I/O object service class, `a` denotes a value of type `X`, `b` denotes a value of type `X::implementation_type`, and `u` denotes an identifier.

Table--IoObjectService requirements

expression	return type	assertion/note pre/post-condition
`X::implementation_type`		
`X::implementation_type u;`		note: `X::implementation_type` has a public default constructor and destructor.
`a.construct(b);`		
`a.destroy(b);`		note: `destroy()` will only be called on a value that has previously been initialised with `construct()`.

5.3.2.5. Thread safety

The following text is intended to document the thread safety guarantees of the library. It needs to be rewritten using some better language.

Unless otherwise specified, all types defined in this clause provide the following basic level of thread safety. For a type `X`:

— Distinct objects `x1` and `x2` of type `X` may be safely used concurrently.

— It is not safe to have concurrent use of a single object `x` of type `X`.

It is safe to have concurrent use of a single object of type `io_service`, `io_service::id`, `io_service::strand` or `io_service::work`.

It is safe to have concurrent use of any `io_service::service`-derived class `X` defined in this clause, provided arguments of type `X::implementation_type` refer to distinct objects.

5.3.2.6. Requirements on synchronous operations

In this clause, a synchronous operation is any function that has a final parameter `error_code& ec`.

Implementations of synchronous operations described in this clause are permitted to call the application programming interface (API) provided by the operating system. If such an operating system API call results in an error, `ec` shall be set to an `error_code` value such that `ec` evaluates to true. Otherwise `ec` shall be set such that `ec` evaluates to false.

Unless otherwise noted, when the behaviour of a synchronous operation is defined "as if" implemented by a *POSIX* function, `ec` shall be set to the errno error or `getaddrinfo` error value of type `error_code` that corresponds to the failure condition described by *POSIX* for that function, if any. Otherwise `ec` shall be set to an implementation-defined `error_code` value that reflects the operating system error.

Except for `io_service::run()` and `io_service::run_one()`, synchronous operations shall not fail with an error condition that indicates interruption by a signal (*POSIX* `EINTR`). [*Note:* Implementations may meet this requirement by automatically restarting the operation if a signal occurs. *—end note*]

Synchronous operations shall not fail with any error condition associated with non-blocking operations (*POSIX* `EWOULDBLOCK`, `EAGAIN` or `EINPROGRESS`; *Windows* `WSAEWOULDBLOCK` or `WSAEINPROGRESS`).

A synchronous operation may have an overload with parameters that differ only by the absence of `error_code& ec`. If the overload returns `void`, the overload shall behave as if implemented in terms of the synchronous operation as follows:

```
void f(A1 a1, A2 a2, ..., AN aN)
{
  error_code ec;
  f(a1, a2, ..., aN, ec);
  if (ec) throw system_error(ec);
}
```

If the overload returns non-`void`, the overload shall behave as if implemented in terms of the synchronous operation as follows:

```
R f(A1 a1, A2 a2, ..., AN aN)
{
  error_code ec;
  R r = f(a1, a2, ..., aN, ec);
  if (ec) throw system_error(ec);
  return r;
}
```

In this clause, when a synchronous operation has an overload as described above, only the behaviour of the synchronous operation is specified, for brevity.

5.3.2.7. Requirements on asynchronous operations

In this clause, an asynchronous operation is initiated by a function that is named with the prefix `async_`. These functions shall be known as *initiating functions*.

All initiating functions in this clause take a function object meeting handler requirements as the final parameter. These handlers accept as their first parameter an lvalue of type `const error_code`.

Implementations of asynchronous operations described in this clause are permitted to call the application programming interface (API) provided by the operating system. If such an operating system API call results in an error, the handler shall be invoked with a `const error_code` lvalue that evaluates to true. Otherwise the handler shall be invoked with a `const error_code` lvalue that evaluates to false.

Unless otherwise noted, when the behaviour of an asynchronous operation is defined "as if" implemented by a *POSIX* function, the handler shall be invoked with an errno error or `getaddrinfo` error value of type `error_code` that corresponds to the failure condition described by *POSIX* for that function, if any. Otherwise the handler shall be invoked with an implementation-defined `error_code` value that reflects the operating system error.

Asynchronous operations shall not fail with an error condition that indicates interruption by a signal (*POSIX* `EINTR`).

Asynchronous operations shall not fail with any error condition associated with non-blocking operations (*POSIX* `EWOULDBLOCK`, `EAGAIN` or `EINPROGRESS`; *Windows* `WSAEWOULDBLOCK` or `WSAEINPROGRESS`).

All asynchronous operations have an associated `io_service` object. Where the initiating function is a member function, the associated `io_service` is that returned by the `io_service()` member function on the same object. Where the initiating function is not a member function, the associated `io_service` is that returned by the `io_service()` member function of the first argument to the initiating function.

Arguments to initiating functions shall be treated as follows:

— If the parameter is declared as a const reference or by-value, the program is not required to guarantee the validity of the argument after the initiating function completes. The implementation may make copies of the argument, and all copies shall be destroyed no later than immediately after invocation of the handler.

— If the parameter is declared as a non-const reference, const pointer or non-const pointer, the program must guarantee the validity of the argument until the handler is invoked.

The library implementation is only permitted to make calls to an initiating function's arguments' copy constructors or destructors from a thread that satisfies one of the following conditions:

— The thread is executing any member function of the associated `io_service` object.

— The thread is executing the destructor of the associated `io_service` object.

— The thread is executing one of the `io_service` service access functions `use_service`, `add_service` or `has_service`, where the first argument is the associated `io_service` object.

— The thread is executing any member function, constructor or destructor of an object of a class defined in this clause, where the object's `io_service()` member function returns the associated `io_service` object.

— The thread is executing any function defined in this clause, where any argument to the function has an `io_service()` member function that returns the associated `io_service` object.

Implementations may use one or more hidden threads to emulate asynchronous functionality. The above requirements are intended to prevent these hidden threads from making calls to program code. This means that a program can, for example, use thread-unsafe reference counting in handler objects, provided the program ensures that all calls to an `io_service` *and related objects occur from the one thread.*

The `io_service` object associated with an asynchronous operation shall have unfinished work, as if by maintaining the existence of one or more objects of class `io_service::work` constructed using the `io_service`, until immediately after the handler for the asynchronous operation has been invoked.

When an asynchronous operation is complete, the handler for the operation will be invoked as if by:

1. Constructing a bound completion handler `bch` for the handler, as described below.
2. Calling `ios.post(bch)` to schedule the handler for deferred invocation, where `ios` is the associated `io_service`.

[*Note:* This implies that the handler must not be called directly from within the initiating function, even if the asynchronous operation completes immediately. *—end note*]

A bound completion handler is a handler object that contains a copy of a user-supplied handler, where the user-supplied handler accepts one or more arguments. The bound completion handler does not accept any arguments, and contains values to be passed as arguments to the user-supplied handler. The bound completion handler forwards the `io_handler_allocate()`, `io_handler_deallocate()`, and `io_handler_invoke()` calls to the corresponding functions for the user-supplied handler. A bound completion handler meets the requirements for a completion handler.

[*Example:* A bound completion handler for a `ReadHandler`:

```
template<class ReadHandler>
struct bound_read_handler
{
  bound_read_handler(ReadHandler handler, const error_code& ec, size_t s)
    : handler_(handler), ec_(ec), s_(s)
  {
  }

  void operator()()
  {
    handler_(ec_, s_);
  }

  ReadHandler handler_;
  const error_code ec_;
  const size_t s_;
};
```

```
template<class ReadHandler>
void* io_handler_allocate(size_t size,
                          bound_read_handler<ReadHandler>* this_handler)
{
  using namespace std::tr2::sys;
  return io_handler_allocate(size, &this_handler->handler_);
}

template<class ReadHandler>
void io_handler_deallocate(void* pointer, std::size_t size,
                           bound_read_handler<ReadHandler>* this_handler)
{
  using namespace std::tr2::sys;
  io_handler_deallocate(pointer, size, &this_handler->handler_);
}

template<class F, class ReadHandler>
void io_handler_invoke(const F& f,
                       bound_read_handler<ReadHandler>* this_handler)
{
  using namespace std::tr2::sys;
  io_handler_invoke(f, &this_handler->handler_);
}
```

—*end example*]

If the thread that initiates an asynchronous operation terminates before the associated handler is invoked, the behaviour is implementation-defined.

The handler argument to an initiating function defines a handler identity. That is, the original handler argument and any copies of the handler argument shall be considered equivalent in the discussion below.

If the implementation needs to allocate storage for an asynchronous operation, the implementation shall perform `io_handler_allocate(size, &h)`, where `size` is the required size in bytes, and `h` is the handler. The implementation shall perform `io_handler_deallocate(p, size, &h)`, where `p` is a pointer to the storage, to deallocate the storage prior to the invocation of the handler via `io_handler_invoke`. Multiple storage blocks may be allocated for a single asynchronous operation.

The implementation shall not make concurrent calls to `io_handler_allocate`, `io_handler_deallocate` or `io_handler_invoke` for a single handler identity. Calls to `io_handler_allocate`, `io_handler_deallocate` or `io_handler_invoke` shall only be made from threads that satisfy one of the conditions listed above for calls to the initiating function's arguments' copy constructors and destructors.

The following text is intended to describe the behaviour of the implementation with respect to memory visibility. It needs to be rewritten using some better language.

All initiating functions shall include a release memory barrier prior to any effect that may be visible to a different thread.

A call to `io_handler_invoke` shall be preceded by an acquire memory barrier and succeeded by a release memory barrier.

Calls to `io_handler_allocate` or `io_handler_deallocate` from the initiating function shall occur prior to the release memory barrier.

Calls to `io_handler_allocate` or `io_handler_deallocate` that occur immediately before `io_handler_invoke` shall occur after the acquire memory barrier.

Calls to `io_handler_allocate` and `io_handler_deallocate` not from the initiating function and not immediately before `io_handler_invoke` shall be preceded by an acquire memory barrier and succeeded by a release memory barrier.

5.3.3. Class `io_service`

```
namespace std {
  namespace tr2 {
    namespace sys {

      class io_service
      {
      public:
        // types:
        class service;
        class id;
        class work;
        class strand;
```

```
        // constructors/destructor:
        io_service();
        ~io_service(); // non-virtual

        // members:
        size_t run();
        size_t run(error_code& ec);

        size_t run_one();
        size_t run_one(error_code& ec);

        size_t poll();
        size_t poll(error_code& ec);

        size_t poll_one();
        size_t poll_one(error_code& ec);

        void stop();

        void reset();

        template<class CompletionHandler>
          void dispatch(CompletionHandler handler);

        template<class CompletionHandler>
          void post(CompletionHandler handler);

        template<class Handler>
          unspecified wrap(Handler handler);

      private:
        io_service(const io_service&);     // not defined
        void operator=(const io_service&); // not defined
      };

      // service access:
      template<class Service> Service& use_service(io_service& ios);
      template<class Service> void add_service(io_service& ios, Service* svc);
      template<class Service> bool has_service(io_service& ios) const;
      class service_already_exists : public logic_error { ... };
      class invalid_service_owner : public logic_error { ... };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

Class `io_service` implements an extensible, type-safe, polymorphic set of I/O services, indexed by service *type*. An object of class `io_service` must be initialised before I/O objects such as sockets, resolvers and timers can be used. These I/O objects are distinguished by having constructors that accept an `io_service&` parameter.

Access to the services of an `io_service` is via three function templates, `use_service<>`, `add_service<>` and `has_service<>`.

In a call to `use_service<Service>()`, the type argument chooses a service, making available all members of the named type. If `Service` is not present in an `io_service`, an object of type `Service` is created and added to the io_service. A C++ program can check if an `io_service` implements a particular service with the function template `has_service<Service>()`.

Service objects may be explicitly added to an `io_service` using the function template `add_service<Service>()`. If the `Service` is already present, the `service_already_exists` exception is thrown. If the owner of the service is not the same object as the `io_service` parameter, the `invalid_service_owner` exception is thrown.

Once a service reference is obtained from an `io_service` object by calling `use_service<>`, that reference remains usable as long as the owning `io_service` object exists.

Synchronous operations on I/O objects implicitly run the `io_service` object for an individual operation. The `io_service` functions `run()`, `run_one()`, `poll()` or `poll_one()` must be called for the `io_service` to perform asynchronous operations on behalf of a C++ program. Notification that an asynchronous operation has completed is delivered by invocation of the associated handler. Handlers are invoked only by a thread that is currently calling any overload of `run()`, `run_one()`, `poll()` or `poll_one()` for the `io_service`.

5.3.3.1. `io_service` constructors/destructor

`io_service();`

Effects: Creates an object of class `io_service`.

`~io_service();`

Effects: Destroys an object of class `io_service`. For each service object `svc` in the `io_service` set, in reverse order of the beginning of service object lifetime (C++ Std, 3.8), performs `svc->shutdown_service()`. Then, uninvoked handler objects that were scheduled for deferred invocation on the `io_service`, or any associated `strand`, are destroyed. Then, for each service object `svc` in the `io_service` set, in reverse order of the beginning of service object lifetime, performs `delete static_cast<io_service::service*>(svc)`.

The destruction sequence described above permits programs to simplify their resource management by using `shared_ptr<>`. Where an object's lifetime is tied to the lifetime of a connection (or some other sequence of asynchronous operations), a `shared_ptr` to the object would be bound into the handlers for all asynchronous operations associated with it. This works as follows:

— When a single connection ends, all associated asynchronous operations complete. The corresponding handler objects are destroyed, and all `shared_ptr` references to the objects are destroyed.

— To shut down the whole program, the `io_service` function `stop()` is called to terminate any `run()` calls as soon as possible. The `io_service` destructor defined above destroys all handlers, causing all `shared_ptr` references to all connection objects to be destroyed.

5.3.3.2. `io_service` members

```
size_t run();
size_t run(error_code& ec);
```

Requires: Must not be called from a thread that is currently calling one of `run()`, `run_one()`, `poll()`, or `poll_one()`.

Effects: Performs `io_service` work until there is no more work to do, or the `io_service` is explicitly stopped, as if implemented as follows:

```
size_t n = 0;
while (run_one(ec))
  if (n != numeric_limits<size_t>::max())
    ++n;
```

Returns: `n`.

```
size_t run_one();
size_t run_one(error_code& ec);
```

Requires: Must not be called from a thread that is currently calling one of `run()`, `run_one()`, `poll()`, or `poll_one()`.

Effects: Does not return until one of the following conditions is true:

— An error has occurred. `ec` shall be set to the `error_code` value corresponding to the failure condition.

— The `io_service` has been explicitly stopped by a call to `stop()`. `ec` shall be set such that the expression `!ec` is true.

— No object `w` of class `io_service::work` exists where `&w.io_service() == this`. The `io_service` shall be stopped as if by calling `stop()` and `ec` shall be set such that the expression `!ec` is true.

— One handler `h` has been invoked by performing `io_handler_invoke(h, &h)`. `ec` shall be set such that the expression `!ec` is true.

If the function can complete due to more than one of these conditions, the earliest condition listed is chosen, and the effects associated with the other conditions shall not be performed.

If the invoked handler throws an exception, the exception shall be allowed to propagate to the caller of `run_one()`. The `io_service` state shall be equivalent to if the handler had been successfully invoked without throwing an exception.

Returns: `1` if a handler was invoked, otherwise `0`.

Notes: This function may invoke additional handlers through nested calls to `dispatch()`. These do not count towards the return value.

```
size_t poll();
size_t poll(error_code& ec);
```

Effects: Performs `io_service` work without blocking until there are no handlers ready to be dispatched immediately, or the `io_service` is stopped, as if implemented as follows:

```
size_t n = 0;
while (poll_one(ec))
  if (n != numeric_limits<size_t>::max())
    ++n;
```

Returns: `n`.

```
size_t poll_one();
size_t poll_one(error_code& ec);
```

Effects: Performs one of the following actions:

— If an error has occurred, sets `ec` to any `error_code` value such that the expression `!ec` is false.

— If the `io_service` has been explicitly stopped by a call to `stop()`, sets `ec` shall such that the expression `!ec` is true.

— If no object `w` of class `io_service::work` exists where `&w.io_service() == this`, stops the `io_service` as if by calling `stop()` and sets `ec` such that the expression `!ec` is true.

— If a handler `h` is available for immediate invocation, invokes the handler by performing `io_handler_invoke(h, &h)`, and sets `ec` such that the expression `!ec` is true.

— Otherwise, sets `ec` such that the expression `!ec` is true.

If the function can succeed in more than one of these ways, the earliest condition listed is chosen, and the effects associated with the other conditions shall not be performed.

If the invoked handler throws an exception, the exception shall be allowed to propagate to the caller of `poll_one()`. The `io_service` state shall be equivalent to if the handler had been successfully invoked without throwing an exception.

Returns: `1` if a handler was invoked, otherwise `0`.

Notes: This function may invoke additional handlers through nested calls to `dispatch()`. These do not count towards the return value.

```
void stop();
```

Effects: Signals the `io_service` to enter the stopped state. Concurrent calls to any overload of `run()`, `run_one()`, `poll()` or `poll_one()` will end as soon as possible; calls to `run()`, `run_one()`, `poll()` or `poll_one()` that are currently dispatching a handler will end only after completion of that handler. The `stop()` call returns without waiting for concurrent calls to `run()`, `run_one()`, `poll()` or `poll_one()` to exit. While an `io_service` is in the stopped state, calls to `run()`, `run_one()`, `poll()` or `poll_one()` will exit immediately with a return value of `0`, without dispatching any handlers. An `io_service` remains in the stopped state until a call to `reset()`.

```
void reset();
```

Requires: There must be no concurrent call to any overload of `run()`, `run_one()`, `poll()`, `poll_one()` or `stop()`.

Effects: Signals the `io_service` to leave the stopped state which was entered as an effect of a prior call to `stop()`.

```
template<class CompletionHandler>
  void dispatch(CompletionHandler handler);
```

Effects: If the current thread is executing a call to any overload of `run()`, `run_one()`, `poll()` or `poll_one()` for the same `io_service` object, invokes the completion handler by performing `io_handler_invoke(handler, &handler)`. Otherwise, equivalent to calling `post(handler)`.

If the invoked handler throws an exception, the exception shall be allowed to propagate to the caller of `dispatch()`. The `io_service` state shall be equivalent to if the handler had been successfully invoked without throwing an exception.

```
template<class CompletionHandler>
  void post(CompletionHandler handler);
```

Effects: Requests invocation of the handler by the `io_service`. Does not invoke the handler from within the call to `post()`. The `io_service` makes a copy of the handler and maintains one or more copies of the handler until immediately after the handler has been invoked. The `io_service` shall have unfinished work, as if by maintaining the existence of one or more objects of class `io_service::work` constructed using the `io_service` object `*this`, until immediately after the handler has been invoked.

The `post()` function behaves according to the rules for asynchronous operations with respect to the treatment of `handler` as a by-value parameter, storage allocation and memory visibility. The object `*this` is considered to be the associated `io_service`.

```
template<class Handler>
  unspecified wrap(Handler handler);
```

Returns: An object `f` of an unspecified type `F` meeting handler requirements, and constructed using `F(*this, handler)`, where `F` behaves as if defined as follows:

```
class F
{
public:
  F(io_service& i, Handler h)
    : io_service_(i), handler_(h) {}

  template<class T1, class T2, ..., class TN>
  class GN
  {
  public:
    GN(Handler h, T1 t1, T2 t2, ..., TN tN)
      : handler_(h), t1_(t1), t2_(t2), ..., tN_(tN) {}

    void operator()() { handler_(t1_, t2_, ..., tN_); }

    friend void* io_handler_allocate(size_t size, GN* this_g)
    {
      return io_handler_allocate(size, &this_g->handler_);
    }

    friend void io_handler_deallocate(void* pointer, size_t size, GN* this_g)
    {
      io_handler_deallocate(pointer, size, &this_g->handler_);
    }

    template<class Function>
    friend void io_handler_invoke(Function func, GN* this_g)
    {
      io_handler_invoke(func, &this_g->handler_);
    }

  private:
    Handler handler_;
    T1 t1_; T2 t2_; ...; TN tN_;
  };

  template<class T1, class T2, ..., class TN>
  void operator()(T1 t1, T2 t2, ..., TN tN)
  {
    io_service_.dispatch(GN<T1,T2,...,TN>(handler_, t1, t2, ..., tN));
  }

  friend void* io_handler_allocate(size_t size, F* this_f)
  {
    return io_handler_allocate(size, &this_f->handler_);
  }

  friend void io_handler_deallocate(void* pointer, size_t size, F* this_f)
  {
    io_handler_deallocate(pointer, size, &this_f->handler_);
```

```
  }

  template<class Function>
  class H
  {
  public:
    H(Function f, Handler h)
      : function_(f), handler_(h) {}

    void operator()()
    {
      function_();
    }

    friend void* io_handler_allocate(size_t size, H* this_h)
    {
      return io_handler_allocate(size, &this_h->handler_);
    }

    friend void io_handler_deallocate(void* pointer, size_t size, H* this_h)
    {
      io_handler_deallocate(pointer, size, &this_h->handler_);
    }

    template<class Function1>
    friend void io_handler_invoke(Function1 func, H* this_h)
    {
      io_handler_invoke(func, &this_h->handler_);
    }

  private:
    Function function_;
    Handler handler_;
  };

  template<class Function>
  friend void io_handler_invoke(Function func, F* this_f)
  {
    this_f->io_service_.dispatch(H<Function>(func, this_f->handler_));
  }

private:
  io_service& io_service_;
  Handler handler_;
};
```

[*Note:* In practical terms, the effect of `f(v1, v2, ..., vN)` is to cause the `io_service` to perform `handler(v1, v2, ..., vN)` from within a call to `run()`, `run_one()`, `poll()` or `poll_one()`.

Similarly, the effect of `io_handler_invoke(g, &f)` is to cause the `io_service` to perform `g()` from within a call to `run()`, `run_one()`, `poll()` or `poll_one()`. *—end note*]

5.3.3.3. `io_service` globals

`template<class Service> Service& use_service(io_service& ios);`

Requires: `Service` is a service class whose definition contains the public static member `id` as defined here.

Effects: If an object of type `Service` does not already exist in the `io_service` set identified by `ios`, creates an object as if by calling `new Service(ios)` and adds it to the set.

Returns: A reference to the corresponding service of `ios`.

Notes: The reference returned remains valid as long as the `io_service` object `ios` exists.

`template<class Service> void add_service(io_service& ios, Service* svc);`

Requires: `Service` is a service class whose definition contains the public static member `id` as defined here. The argument `svc` is a non-null pointer to a service object, and the condition `&svc->io_service() == &ios` is true. A corresponding service object does not already exist in the `io_service` set identified by `ios`.

Effects: Adds the service object `svc` to the `io_service` set.

Throws: `service_already_exists` if a corresponding service object is already present in the `io_service` set identified by `ios`; `invalid_service_owner` if the condition `&svc->io_service() == &ios` is false.

```
template<class Service> bool has_service(io_service& ios) const;
```

Requires: `Service` is a service class whose definition contains the public static member `id` as defined here.

Returns: `true` if the service requested is present in `ios`, otherwise `false`.

5.3.4. Class `io_service::service`

```
namespace std {
  namespace tr2 {
    namespace sys {

      class io_service::service
      {
      public:
        std::tr2::sys::io_service& io_service();

      protected:
        service(std::tr2::sys::io_service& owner);
        virtual ~service();

      private:
//      friend class io_service;  exposition only

        virtual void shutdown_service() = 0;

        service(const service&);        // not defined
        void operator=(const service&); // not defined
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.3.5. Class `io_service::id`

```
namespace std {
  namespace tr2 {
    namespace sys {

      class io_service::id
      {
      public:
        id();

      private:
        id(const id&);             // not defined
        void operator=(const id&); // not defined
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

The class `io_service::id` provides identification of services, and is used as an index for service lookup.

5.3.6. Class `io_service::work`

```
namespace std {
  namespace tr2 {
    namespace sys {

      class io_service::work
      {
      public:
        // constructors/destructor:
        explicit work(std::tr2::sys::io_service& ios);
        work(const work& other);
        ~work();

        // members:
        std::tr2::sys::io_service& io_service();
```

```
      private:
        void operator=(const work&); // not defined
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

An object of class `io_service::work` represents a unit of unfinished work for an `io_service`.

5.3.6.1. `io_service::work` constructors/destructor

`explicit work(std::tr2::sys::io_service& ios);`

> *Effects:* Constructs an object of class `work`.
>
> *Postconditions:* `&io_service() == &ios`.

`work(const work& other);`

> *Effects:* Constructs an object of class `work`.
>
> *Postconditions:* `&io_service() == &other.io_service()`.

`~work();`

> *Effects:* If no other `work` object `w` exists such that `&io_service() == &w.io_service()`, stops the `io_service` as if by calling `io_service().stop()`.

5.3.6.2. `io_service::work` members

`std::tr2::sys::io_service& io_service();`

> *Returns:* The `io_service` associated with the work.

5.3.7. Class `io_service::strand`

```
namespace std {
  namespace tr2 {
    namespace sys {

      class io_service::strand
      {
      public:
        // constructors/destructor:
        explicit strand(std::tr2::sys::io_service& ios);
        ~strand();

        // members:
        std::tr2::sys::io_service& io_service();

        template<class CompletionHandler>
          void dispatch(CompletionHandler handler);

        template<class CompletionHandler>
          void post(CompletionHandler handler);

        template<class Handler>
          unspecified wrap(Handler handler);

      private:
        strand(const strand&);          // not defined
        void operator=(const strand&); // not defined
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

An object of class `io_service::strand` may be used to prevent concurrent invocation of handlers.

Given:

— An `io_service` object `i`.

— A `strand` object `s` constructed using `s(i)`.

— An object `a` meeting completion handler requirements, where `a` is passed to the `strand` by performing either `s.dispatch(a)` or `s.post(a)`. For this discussion `a` and all copies of `a` are considered equivalent.

— An object `b` meeting completion handler requirements, where `b` is passed to the `strand` by performing either `s.dispatch(b)` or `s.post(b)`. For this discussion `b` and all copies of `b` are considered equivalent.

— An arbitrary function object `x` that is valid in the expression `x()`.

— An arbitrary function object `y` that is valid in the expression `y()`.

— An arbitrary thread *thread1* that is calling any overload of `run()`, `run_one()`, `poll()` or `poll_one()` for the `io_service` object `i`.

— An arbitrary thread *thread2* that is calling any overload of `run()`, `run_one()`, `poll()` or `poll_one()` for the `io_service` object `i`.

The implementation shall ensure that, if *thread1* performs `io_handler_dispatch(x, &a)`, *thread2* does not concurrently perform `io_handler_dispatch(y, &b)`.

[*Note:* No requirements are made on implementations with respect to:

— whether handlers dispatched through different `strand` objects may or may not be invoked concurrently; or

— the order of invocation of handler objects dispatched through a `strand`. —*end note*]

5.3.7.1. `io_service::strand` constructors/destructor

```
explicit strand(std::tr2::sys::io_service& ios);
```

Effects: Constructs an object of class `strand`.

Postconditions: `&io_service() == &ios`.

```
~strand();
```

Effects: Destroys an object of class `strand`. Handlers posted through the strand that have not yet been invoked will still be dispatched in a way that meets the guarantee of non-concurrency.

5.3.7.2. `io_service::strand` members

```
std::tr2::sys::io_service& io_service();
```

Returns: The `io_service` associated with the strand.

```
template<class CompletionHandler>
  void dispatch(CompletionHandler handler);
```

Effects: If the current thread is invoking a handler for the `strand` object `*this`, performs `io_handler_invoke(handler, &handler)`. Otherwise, requests invocation of the handler as if by performing `io_service().dispatch(handler)`, with the additional requirement of non-concurrent handler invocation as defined above.

If the invoked handler throws an exception, the exception shall be allowed to propagate to the caller of `dispatch()`. The `io_service` and `strand` states shall be equivalent to if the handler had been successfully invoked without throwing an exception.

```
template<class CompletionHandler>
  void post(CompletionHandler handler);
```

Effects: Requests invocation of the handler as if by performing `io_service().post(handler)`, with the additional requirement of non-concurrent handler invocation as defined above.

```
template<class Handler>
  unspecified wrap(Handler handler);
```

Returns: An object `f` of an unspecified type `F` meeting handler requirements, and constructed using `F(*this, handler)`, where `F` behaves as if defined as follows:

```
class F
{
public:
```

```
F(strand& s, Handler h)
  : strand_(s), handler_(h) {}

template<class T1, class T2, ..., class TN>
class GN
{
public:
  GN(Handler h, T1 t1, T2 t2, ..., TN tN)
    : handler_(h), t1_(t1), t2_(t2), ..., tN_(tN) {}

  void operator()() { handler_(t1_, t2_, ..., tN_); }

  friend void* io_handler_allocate(size_t size, GN* this_g)
  {
    return io_handler_allocate(size, &this_g->handler_);
  }

  friend void io_handler_deallocate(void* pointer, size_t size, GN* this_g)
  {
    io_handler_deallocate(pointer, size, &this_g->handler_);
  }

  template<class Function>
  friend void io_handler_invoke(Function func, GN* this_g)
  {
    io_handler_invoke(func, &this_g->handler_);
  }

private:
  Handler handler_;
  T1 t1_; T2 t2_; ...; TN tN_;
};

template<class T1, class T2, ..., class TN>
void operator()(T1 t1, T2 t2, ..., TN tN)
{
  strand_.dispatch(GN<T1,T2,...,TN>(handler_, t1, t2, ..., tN));
}

friend void* io_handler_allocate(size_t size, F* this_f)
{
  return io_handler_allocate(size, &this_f->handler_);
}

friend void io_handler_deallocate(void* pointer, size_t size, F* this_f)
{
  io_handler_deallocate(pointer, size, &this_f->handler_);
}

template<class Function>
class H
{
public:
  H(Function f, Handler h)
    : function_(f), handler_(h) {}

  void operator()()
  {
    function_();
  }

  friend void* io_handler_allocate(size_t size, H* this_h)
  {
    return io_handler_allocate(size, &this_h->handler_);
  }

  friend void io_handler_deallocate(void* pointer, size_t size, H* this_h)
  {
    io_handler_deallocate(pointer, size, &this_h->handler_);
  }

  template<class Function1>
  friend void io_handler_invoke(Function1 func, H* this_h)
  {
    io_handler_invoke(func, &this_h->handler_);
  }

private:
```

```
      Function function_;
      Handler handler_;
    };

    template<class Function>
    friend void io_handler_invoke(Function func, F* this_f)
    {
      this_f->strand_.dispatch(H<Function>(func, this_f->handler_));
    }

  private:
    strand& strand_;
    Handler handler_;
  };
```

[*Note:* In practical terms, the effect of `f(v1, v2, ..., vN)` is to cause the `io_service` associated with the `strand` to perform `handler(v1, v2, ..., vN)` from within a call to `run()`, `run_one()`, `poll()` or `poll_one()`, subject to the requirements of non-concurrent handler invocation as defined above.

Similarly, the effect of `io_handler_invoke(g, &f)` is to cause the `io_service` to perform `g()` from within a call to `run()`, `run_one()`, `poll()` or `poll_one()`, subject to the requirements of non-concurrent handler invocation as defined above. *—end note*]

5.3.8. Default handler hook functions

```
void* io_handler_allocate(size_t s, ...);
```

Returns: `::operator new(s)`.

```
void io_handler_deallocate(void* p, size_t s, ...);
```

Effects: Calls `::operator delete(p)`.

```
template<class F> void io_handler_invoke(F f, ...);
```

Effects: Performs `f()`.

5.3.9. Class template `basic_io_object`

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class IoObjectService>
      class basic_io_object
      {
      public:
        typedef IoObjectService service_type;
        typedef typename IoObjectService::implementation_type
          implementation_type;

        std::tr2::sys::io_service& io_service();

      protected:
        explicit basic_io_object(std::tr2::sys::io_service& io_service);
        ~basic_io_object();

        service_type& service;
        implementation_type implementation;

      private:
        basic_io_object(const basic_io_object&); // not defined
        void operator=(const basic_io_object&);  // not defined
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.3.9.1. `basic_io_object` members

```
explicit basic_io_object(std::tr2::sys::io_service& io_service);
```

Effects: Calls `service.construct(implementation)`.

```
~basic_io_object();
```

Effects: Calls `service.destroy(implementation)`.

```
std::tr2::sys::io_service& io_service();
```

Returns: `service.io_service()`.

5.4. Timers

This subclause defines components for performing timer operations.

[*Example:* Performing a synchronous wait operation on a timer:

```
io_service i;
deadline_timer t(i);
t.expires_from_now(seconds(5));
t.wait();
```

—*end example*]

[*Example:* Performing an asynchronous wait operation on a timer:

```
void handler(error_code ec) { ... }
...
io_service i;
deadline_timer t(i);
t.expires_from_now(seconds(5));
t.async_wait(handler);
i.run();
```

—*end example*]

5.4.1. Header `<timer>` synopsis

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Time> struct time_traits;

      template<class Time, class TimeTraits = time_traits<Time> >
        class deadline_timer_service;

      template<class Time, class TimeTraits = time_traits<Time>,
        class TimerService = deadline_timer_service<Time, TimeTraits> >
          class basic_deadline_timer;

      typedef basic_deadline_timer<date_time> deadline_timer;

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.4.2. Requirements

5.4.2.1. Time traits requirements

In the table below, `X` denotes a time traits class for time type `Time`, `t`, `t1`, and `t2` denote values of type `Time`, and `d` denotes a value of type `X::duration_type`.

Table--TimeTraits requirements

expression	return type	assertion/note pre/post-condition
`X::time_type`	`Time`	Represents an absolute time. Must support default construction, and meet the requirements for `CopyConstructible` and `Assignable`.
`X::duration_type`		Represents the difference between two absolute times. Must support default construction, and meet the requirements for `CopyConstructible` and `Assignable`. A duration can be positive, negative, or zero.

expression	return type	assertion/note pre/post-condition
`X::now();`	`time_type`	Returns the current time.
`X::add(t, d);`	`time_type`	Returns a new absolute time resulting from adding the duration `d` to the absolute time `t`.
`X::subtract(t1, t2);`	`duration_type`	Returns the duration resulting from subtracting `t2` from `t1`.
`X::less_than(t1, t2);`	`bool`	Returns whether `t1` is to be treated as less than `t2`.
`X::to_std_duration(d);`	`date_time::time_duration_type`	Returns the `date_time::time_duration_type` value that most closely represents the duration `d`.

5.4.2.2. Wait handler requirements

A wait handler must meet the requirements for a handler. A value `h` of a wait handler class should work correctly in the expression `h(ec)`, where `ec` is an lvalue of type `const error_code`.

5.4.2.3. Timer service requirements

A timer service must meet the requirements for an I/O object service, as well as the additional requirements listed below.

In the table below, `X` denotes a timer service class for time type `Time` and traits type `TimeTraits`, `a` denotes a value of type `X`, `b` denotes a value of type `X::implementation_type`, `t` denotes a value of type `Time`, `d` denotes a value of type `TimeTraits::duration_type`, `e` denotes a value of type `error_code`, and `h` denotes a value meeting `WaitHandler` requirements.

Table--TimerService requirements

expression	return type	assertion/note pre/post-condition
`a.destroy(b);`		From IoObjectService requirements. Implicitly cancels asynchronous wait operations, as if by calling `a.cancel(b, e)`.
`a.cancel(b, e);`	`size_t`	Causes any outstanding asynchronous wait operations to complete as soon as possible. Handlers for cancelled operations shall be passed the error code `error::operation_aborted`. Sets `e` to indicate success or failure. Returns the number of operations that were cancelled.
`a.expires_at(b);`	`Time`	
`a.expires_at(b, t, e);`	`size_t`	Implicitly cancels asynchronous wait operations, as if by calling `a.cancel(b, e)`. Returns the number of operations that were cancelled. post: `a.expires_at(b) == t`.
`a.expires_from_now(b);`	`TimeTraits::duration_type`	Returns a value equivalent to `TimeTraits::subtract(a.expires_at(b), TimeTraits::now())`.
`a.expires_from_now(b, d, e);`	`size_t`	Equivalent to `a.expires_at(b, TimeTraits::add(TimeTraits::now(), d), e)`.
`a.wait(b, e);`	`error_code`	Sets `e` to indicate success or failure. Returns `e`. post: `!!e || !TimeTraits::lt(TimeTraits::now(), a.expires_at(b))`.
`a.async_wait(b, h);`		Initiates an asynchronous wait operation that is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. The handler shall be posted for execution only if the

expression	return type	assertion/note pre/post-condition
		condition `!!ec || !TimeTraits::lt(TimeTraits::now(), a.expires_at(b))` holds, where `ec` is the error code to be passed to the handler.

5.4.3. `time_traits` specialisations

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<> struct time_traits<date_time>;

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.4.3.1. Struct `time_traits<date_time>`

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<>
      struct time_traits<date_time>
      {
        // types:
        typedef date_time time_type;
        typedef date_time::time_duration duration_type;

        // static members:
        static time_type now();

        static time_type add(const time_type& t, const duration_type& d);

        static duration_type subtract(const time_type& t1,
                                      const time_type& t2);

        static bool less_than(const time_type& t1, const time_type& t2);

        static date_time::time_duration to_std_duration(
          const duration_type& d);
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.4.3.1.1. `time_traits<date_time>` static members

```
static time_type now();
```

Returns: `microsecond_clock::universal_time()`.

```
static time_type add(const time_type& t, const duration_type& d);
```

Returns: `t + d`.

```
static duration_type subtract(const time_type& t1, const time_type& t2);
```

Returns: `t1 - t2`.

```
static bool less_than(const time_type& t1, const time_type& t2);
```

Returns: `t1 < t2`.

```
static date_time::time_duration to_std_duration(const duration_type& d);
```

Returns: `d`.

5.4.4. Class template deadline_timer_service

Instances of the deadline_timer_service class template meet the requirements of a TimerService.

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Time, class TimeTraits>
      class deadline_timer_service :
        public io_service::service
      {
      public:
        static io_service::id id;

        // types:
        typedef TimeTraits traits_type;
        typedef Time time_type;
        typedef typename TimeTraits::duration_type duration_type;
        typedef unspecified implementation_type;

        // constructors:
        explicit deadline_timer_service(io_service& ios);

        // members:
        void construct(implementation_type& impl);

        void destroy(implementation_type& impl);

        size_t cancel(implementation_type& impl, error_code& ec);

        time_type expires_at(const implementation_type& impl) const;

        size_t expires_at(implementation_type& impl, const time_type& t,
                          error_code& ec);

        duration_type expires_from_now(const implementation_type& impl) const;

        size_t expires_from_now(implementation_type& impl,
                                const duration_type& d, error_code& ec);

        error_code wait(implementation_type& impl, error_code& ec);

        template<class WaitHandler>
          void async_wait(implementation_type& impl, WaitHandler handler);

      private:
        virtual void shutdown_service();
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.4.4.1. deadline_timer_service constructors

```
explicit deadline_timer_service(io_service& ios);
```

Effects: Constructs an object of class `deadline_timer_service<Time, TimeTraits>`, initialising the base class with `io_service::service(ios)`.

5.4.4.2. deadline_timer_service members

```
void shutdown_service();
```

Effects: Destroys all copies of user-defined handler objects owned by the service.

```
void construct(implementation_type& impl);
```

Effects: Initialises the timer implementation `impl`.

```
void destroy(implementation_type& impl);
```

Effects: Cleans up resources owned by the timer implementation `impl`. Cancels asynchronous wait operations associated with `impl` as if by performing:

```
error_code ec;
cancel(impl, ec);
```

```
size_t cancel(implementation_type& impl, error_code& ec);
```

Effects: Causes any outstanding asynchronous wait operations to complete as soon as possible. Handlers for cancelled operations shall be passed the error code `error::operation_aborted`.

Returns: The number of operations that were cancelled.

```
time_type expires_at(const implementation_type& impl) const;
```

Returns: The expiry time associated with the timer implementation `impl`, as previously set using `expires_at()` or `expires_from_now()`.

```
size_t expires_at(implementation_type& impl, const time_type& t,
                  error_code& ec);
```

Effects: Sets the expiry time associated with the timer implementation `impl`. Implicitly cancels asynchronous wait operations, as if by calling `cancel(impl, ec)`.

Returns: The number of operations that were cancelled.

Postconditions: `expires_at(impl) == t`.

```
duration_type expires_from_now(const implementation_type& impl) const;
```

Returns: A value equivalent to `TimeTraits::subtract(a.expires_at(b), TimeTraits::now())`.

```
size_t expires_from_now(implementation_type& impl,
                        const duration_type& d, error_code& ec);
```

Effects: Equivalent to `a.expires_at(b, TimeTraits::add(TimeTraits::now(), d), ec)`.

```
error_code wait(implementation_type& impl, error_code& ec);
```

Effects: Performs a synchronous wait operation associated with the timer implementation `impl`.

Returns: `ec`.

Postconditions: `!!ec || !TimeTraits::lt(TimeTraits::now(), a.expires_at(b))`.

```
template<class WaitHandler>
  void async_wait(implementation_type& impl, WaitHandler handler);
```

Effects: Initiates an asynchronous wait operation that is performed via the `io_service` object returned by `io_service()` and behaves according to asynchronous operation requirements.

The handler shall be posted for invocation only if the condition `!!ec || !TimeTraits::lt(TimeTraits::now(), expires_at(impl))` holds, where `ec` is the error code to be passed to the handler.

5.4.5. Class template `basic_deadline_timer`

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Time, class TimeTraits, class TimerService>
      class basic_deadline_timer :
        public basic_io_object<ResolverService>
      {
      public:
        // types:
        typedef TimeTraits traits_type;
        typedef Time time_type;
        typedef typename TimeTraits::duration_type duration_type;

        // constructors:
        explicit basic_deadline_timer(std::tr2::sys::io_service& io_service);
        basic_deadline_timer(std::tr2::sys::io_service& io_service,
                             const time_type& t);
        basic_deadline_timer(std::tr2::sys::io_service& io_service,
                             const duration_type& d);
```

```
      // members:
      size_t cancel();
      size_t cancel(error_code& ec);

      time_type expires_at() const;

      size_t expires_at(const time_type& t);
      size_t expires_at(const time_type& t, error_code& ec);

      duration_type expires_from_now() const;

      size_t expires_from_now(const duration_type& d);
      size_t expires_from_now(const duration_type& d, error_code& ec);

      void wait();
      error_code wait(error_code& ec);

      template <class WaitHandler>
        void async_wait(WaitHandler handler);
    };

  } // namespace sys
 } // namespace tr2
} // namespace std
```

5.4.5.1. `basic_deadline_timer` constructors

```
explicit basic_deadline_timer(std::tr2::sys::io_service& io_service);
```

Effects: Constructs an object of class `basic_deadline_timer<Time, TimeTraits, TimerService>`, initialising the base class with `basic_io_object(io_service)`.

```
basic_deadline_timer(std::tr2::sys::io_service& io_service,
                     const time_type& t);
```

Effects: Constructs an object of class `basic_deadline_timer<Time, TimeTraits, TimerService>`, initialising the base class with `basic_io_object(io_service)`, then setting the expiry time as if by calling:

```
error_code ec;
this->service.expires_at(this->implementation, t, ec);
if (ec) throw system_error(ec);
```

```
basic_deadline_timer(std::tr2::sys::io_service& io_service,
                     const duration_type& d);
```

Effects: Constructs an object of class `basic_deadline_timer<Time, TimeTraits, TimerService>`, initialising the base class with `basic_io_object(io_service)`, then setting the expiry time as if by calling:

```
error_code ec;
this->service.expires_from_now(this->implementation, d, ec);
if (ec) throw system_error(ec);
```

5.4.5.2. `basic_deadline_timer` members

```
size_t cancel();
size_t cancel(error_code& ec);
```

Returns: `this->service.cancel(this->implementation, ec)`.

```
time_type expires_at() const;
```

Returns: `this->service.expires_at(this->implementation)`.

```
size_t expires_at(const time_type& t);
size_t expires_at(const time_type& t, error_code& ec);
```

Returns: `this->service.expires_at(this->implementation, t, ec)`.

```
duration_type expires_from_now() const;
```

Returns: `this->service.expires_from_now(this->implementation)`.

```
size_t expires_from_now(const duration_type& d);
size_t expires_from_now(const duration_type& d, error_code& ec);
```

Returns: `this->service.expires_from_now(this->implementation, d, ec)`.

```
void wait();
error_code wait(error_code& ec);
```

Returns: `this->service.wait(this->implementation, ec)`.

```
template <class WaitHandler>
  void async_wait(WaitHandler handler);
```

Effects: Calls `this->service.async_wait(this->implementation, handler)`.

5.5. Buffers

5.5.1. Header `<buffer>` synopsis

```
namespace std {
  namespace tr2 {
    namespace sys {

      class mutable_buffer;

      template<class T> T buffer_cast(const mutable_buffer&);
      size_t buffer_size(const mutable_buffer&);
      mutable_buffer operator+(const mutable_buffer&, size_t);
      mutable_buffer operator+(size_t, const mutable_buffer&);

      class const_buffer;

      template<class T> T buffer_cast(const const_buffer&);
      size_t buffer_size(const const_buffer&);
      const_buffer operator+(const const_buffer&, size_t);
      const_buffer operator+(size_t, const const_buffer&);

      class mutable_buffers_1;

      class const_buffers_1;

      // buffer creation functions:

      mutable_buffers_1 buffer(void*, size_t);
      const_buffers_1 buffer(const void*, size_t);

      mutable_buffers_1 buffer(const mutable_buffer&);
      mutable_buffers_1 buffer(const mutable_buffer&, size_t);
      const_buffers_1 buffer(const const_buffer&);
      const_buffers_1 buffer(const const_buffer&, size_t);

      template<class T, size_t N>
        mutable_buffers_1 buffer(T (&)[N]);
      template<class T, size_t N>
        mutable_buffers_1 buffer(T (&)[N], size_t);
      template<class T, size_t N>
        const_buffers_1 buffer(const T (&)[N]);
      template<class T, size_t N>
        const_buffers_1 buffer(const T (&)[N], size_t);

      template<class T, size_t N>
        mutable_buffers_1 buffer(array<T, N>&);
      template<class T, size_t N>
        mutable_buffers_1 buffer(array<T, N>&, size_t);
      template<class T, size_t N>
        const_buffers_1 buffer(array<const T, N>&);
      template<class T, size_t N>
        const_buffers_1 buffer(array<const T, N>&, size_t);
      template<class T, size_t N>
        const_buffers_1 buffer(const array<T, N>&);
      template<class T, size_t N>
        const_buffers_1 buffer(const array<T, N>&, size_t);

      template<class T, class Allocator>
        mutable_buffers_1 buffer(vector<T, Allocator>&);
      template<class T, class Allocator>
        mutable_buffers_1 buffer(vector<T, Allocator>&, size_t);
      template<class T, class Allocator>
```

```
  const_buffers_1 buffer(const vector<T, Allocator>&);
template<class T, class Allocator>
  const_buffers_1 buffer(const vector<T, Allocator>&, size_t);

template<class CharT, class Traits, class Allocator>
  const_buffers_1 buffer(const basic_string<CharT, Traits, Allocator>&);
template<class CharT, class Traits, class Allocator>
  const_buffers_1 buffer(const basic_string<CharT, Traits, Allocator>&,
                         size_t);

template<class Allocator = std::allocator<char> >
  class basic_fifobuf;

typedef basic_fifobuf<> fifobuf;

class transfer_all;
class transfer_at_least;

// synchronous read operations:

template<class SyncReadStream, class MutableBufferSequence>
  size_t read(SyncReadStream& stream,
              const MutableBufferSequence& buffers);
template<class SyncReadStream, class MutableBufferSequence>
  size_t read(SyncReadStream& stream,
              const MutableBufferSequence& buffers, error_code& ec);
template<class SyncReadStream, class MutableBufferSequence,
  class CompletionCondition>
    size_t read(SyncReadStream& stream,
                const MutableBufferSequence& buffers,
                CompletionCondition completion_condition);
template<class SyncReadStream, class MutableBufferSequence,
  class CompletionCondition>
    size_t read(SyncReadStream& stream,
                const MutableBufferSequence& buffers,
                CompletionCondition completion_condition,
                error_code& ec);
template<class SyncReadStream, class Allocator>
  size_t read(SyncReadStream& stream, basic_fifobuf<Allocator>& fb);
template<class SyncReadStream, class Allocator>
  size_t read(SyncReadStream& stream, basic_fifobuf<Allocator>& fb,
              error_code& ec);
template<class SyncReadStream, class Allocator,
  class CompletionCondition>
    size_t read(SyncReadStream& stream, basic_fifobuf<Allocator>& fb,
                CompletionCondition completion_condition);
template<class SyncReadStream, class Allocator,
  class CompletionCondition>
    size_t read(SyncReadStream& stream, basic_fifobuf<Allocator>& fb,
                CompletionCondition completion_condition,
                error_code& ec);

// asynchronous read operations:

template<class AsyncReadStream, class MutableBufferSequence,
  class ReadHandler>
    size_t async_read(AsyncReadStream& stream,
                      const MutableBufferSequence& buffers,
                      ReadHandler handler);
template<class AsyncReadStream, class MutableBufferSequence,
  class CompletionCondition, class ReadHandler>
    size_t async_read(AsyncReadStream& stream,
                      const MutableBufferSequence& buffers,
                      CompletionCondition completion_condition,
                      ReadHandler handler);
template<class AsyncReadStream, class Allocator, class ReadHandler>
  size_t async_read(AsyncReadStream& stream,
                    basic_fifobuf<Allocator>& fb,
                    ReadHandler handler);
template<class AsyncReadStream, class Allocator,
  class CompletionCondition, class ReadHandler>
    size_t async_read(AsyncReadStream& stream,
                      basic_fifobuf<Allocator>& fb,
                      CompletionCondition completion_condition,
                      ReadHandler handler);

// synchronous write operations:
```

```
template<class SyncWriteStream, class ConstBufferSequence>
  size_t write(SyncWriteStream& stream,
               const ConstBufferSequence& buffers);
template<class SyncWriteStream, class ConstBufferSequence>
  size_t write(SyncWriteStream& stream,
               const ConstBufferSequence& buffers, error_code& ec);
template<class SyncWriteStream, class ConstBufferSequence,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 const ConstBufferSequence& buffers,
                 CompletionCondition completion_condition);
template<class SyncWriteStream, class ConstBufferSequence,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 const ConstBufferSequence& buffers,
                 CompletionCondition completion_condition,
                 error_code& ec);
template<class SyncWriteStream, class Allocator>
  size_t write(SyncWriteStream& stream, basic_fifobuf<Allocator>& fb);
template<class SyncWriteStream, class Allocator>
  size_t write(SyncWriteStream& stream, basic_fifobuf<Allocator>& fb,
               error_code& ec);
template<class SyncWriteStream, class Allocator,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 basic_fifobuf<Allocator>& fb,
                 CompletionCondition completion_condition);
template<class SyncWriteStream, class Allocator,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 basic_fifobuf<Allocator>& fb,
                 CompletionCondition completion_condition,
                 error_code& ec);

// asynchronous write operations:

template<class AsyncWriteStream, class ConstBufferSequence,
  class WriteHandler>
    size_t async_write(AsyncWriteStream& stream,
                       const ConstBufferSequence& buffers,
                       WriteHandler handler);
template<class AsyncWriteStream, class ConstBufferSequence,
  class CompletionCondition, class WriteHandler>
    size_t async_write(AsyncWriteStream& stream,
                       const ConstBufferSequence& buffers,
                       CompletionCondition completion_condition,
                       WriteHandler handler);
template<class AsyncWriteStream, class Allocator, class WriteHandler>
  size_t async_write(AsyncWriteStream& stream,
                     basic_fifobuf<Allocator>& fb,
                     WriteHandler handler);
template<class AsyncWriteStream, class Allocator,
  class CompletionCondition, class WriteHandler>
    size_t async_write(AsyncWriteStream& stream,
                       basic_fifobuf<Allocator>& fb,
                       CompletionCondition completion_condition,
                       WriteHandler handler);

// synchronous delimited read operations:

template <class SyncReadStream, class Allocator>
  size_t read_until(SyncReadStream& s, basic_fifobuf<Allocator>& fb,
                    char delim);
template <class SyncReadStream, class Allocator>
  size_t read_until(SyncReadStream& s, basic_fifobuf<Allocator>& fb,
                    char delim, error_code& ec);
template <class SyncReadStream, class Allocator>
  size_t read_until(SyncReadStream& s, basic_fifobuf<Allocator>& fb,
                    const string& delim);
template <class SyncReadStream, class Allocator>
  size_t read_until(SyncReadStream& s, basic_fifobuf<Allocator>& fb,
                    const string& delim, error_code& ec);

// asynchronous delimited read operations:

template <class AsyncReadStream, class Allocator, class ReadHandler>
  void async_read_until(AsyncReadStream& s,
                        basic_fifobuf<Allocator>& fb, char delim,
```

```
                              ReadHandler handler);
      template <class AsyncReadStream, class Allocator, class ReadHandler>
        void async_read_until(AsyncReadStream& s,
                              basic_fifobuf<Allocator>& fb,
                              const string& delim, ReadHandler handler);

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.5.2. Requirements

5.5.2.1. Convertible to mutable buffer requirements

A type that meets the requirements for convertibility to a mutable buffer must meet the requirements of `CopyConstructible` types (C++ Std, 20.1.3), and the requirements of `Assignable` types (C++ Std, 23.1).

In the table below, `X` denotes a class meeting the requirements for convertibility to a mutable buffer, `a` and `b` denote values of type `X`, and `u`, `v` and `w` denote identifiers.

Table--ConvertibleToMutableBuffer requirements

expression	postcondition
`mutable_buffer u(a);` `mutable_buffer v(a);`	`buffer_cast<void*>(u) == buffer_cast<void*>(v)` `&& buffer_size(u) == buffer_size(v)`
`mutable_buffer u(a);` `mutable_buffer v = a;`	`buffer_cast<void*>(u) == buffer_cast<void*>(v)` `&& buffer_size(u) == buffer_size(v)`
`mutable_buffer u(a);` `mutable_buffer v; v = a;`	`buffer_cast<void*>(u) == buffer_cast<void*>(v)` `&& buffer_size(u) == buffer_size(v)`

`mutable_buffer u(a);`
`const X& v = a;`
`mutable_buffer w(v);`
	`buffer_cast<void*>(u) == buffer_cast<void*>(w)`
	`&& buffer_size(u) == buffer_size(w)`

`mutable_buffer u(a);`
`X v(a);`
`mutable_buffer w(v);`
	`buffer_cast<void*>(u) == buffer_cast<void*>(w)`
	`&& buffer_size(u) == buffer_size(w)`

`mutable_buffer u(a);`
`X v = a;`
`mutable_buffer w(v);`
	`buffer_cast<void*>(u) == buffer_cast<void*>(w)`
	`&& buffer_size(u) == buffer_size(w)`

`mutable_buffer u(a);`
`X v(b); v = a;`
`mutable_buffer w(v);`
	`buffer_cast<void*>(u) == buffer_cast<void*>(w)`
	`&& buffer_size(u) == buffer_size(w)`

5.5.2.2. Mutable buffer sequence requirements

In the table below, `X` denotes a class containing objects of type `T`, `a` denotes a value of type `X` and `u` denotes an identifier.

Table--MutableBufferSequence requirements

expression	return type	assertion/note pre/post-condition
`X::value_type`	`T`	`T` meets the requirements for ConvertibleToMutableBuffer.
`X::const_iterator`	iterator type pointing to `T`	`const_iterator` meets the requirements for bidirectional iterators (C++ Std, 24.1.4).

`X(a);`		post: `equal_mutable_buffer_seq(a, X(a))` where the binary predicate `equal_mutable_buffer_seq` is defined as
		`bool equal_mutable_buffer_seq(`
		`  const X& x1, const X& x2)`
		`{`
		`  return`
		`    distance(x1.begin(), x1.end())`
		`      == distance(x2.begin(), x2.end())`
		`        && equal(x1.begin(), x1.end(),`

<table>
<tr><th>expression</th><th>return type</th><th>assertion/note
pre/post-condition</th></tr>
<tr><td></td><td></td><td><pre>   x2.begin(), equal_buffer);
}</pre>
and the binary predicate <code>equal_buffer</code> is defined as
<pre>bool equal_buffer(
  const X::value_type& v1,
  const X::value_type& v2)
{
  mutable_buffer b1(v1);
  mutable_buffer b2(v2);
  return
    buffer_cast<const void*>(b1)
      == buffer_cast<const void*>(b2)
        && buffer_size(b1) == buffer_size(b2);
}</pre></td></tr>
<tr><td><code>X u(a);</code></td><td></td><td>post:
<pre>distance(a.begin(), a.end())
  == distance(u.begin(), u.end())
    && equal(a.begin(), a.end(),
             u.begin(), equal_buffer)</pre>
where the binary predicate <code>equal_buffer</code> is defined as
<pre>bool equal_buffer(
  const X::value_type& v1,
  const X::value_type& v2)
{
  mutable_buffer b1(v1);
  mutable_buffer b2(v2);
  return
    buffer_cast<const void*>(b1)
      == buffer_cast<const void*>(b2)
        && buffer_size(b1) == buffer_size(b2);
}</pre></td></tr>
<tr><td><code>(&a)->~X();</code></td><td><code>void</code></td><td>note: the destructor is applied to every element of <code>a</code>; all the memory is deallocated.</td></tr>
<tr><td><code>a.begin();</code></td><td><code>const_iterator</code> or convertible to <code>const_iterator</code></td><td></td></tr>
<tr><td><code>a.end();</code></td><td><code>const_iterator</code> or convertible to <code>const_iterator</code></td><td></td></tr>
</table>

5.5.2.3. Convertible to const buffer requirements

A type that meets the requirements for convertibility to a const buffer must meet the requirements of `CopyConstructible` types (C++ Std, 20.1.3), and the requirements of `Assignable` types (C++ Std, 23.1).

In the table below, `X` denotes a class meeting the requirements for convertibility to a const buffer, `a` and `b` denote values of type `X`, and `u`, `v` and `w` denote identifiers.

Table--ConvertibleToConstBuffer requirements

<table>
<tr><th>expression</th><th>postcondition</th></tr>
<tr><td><pre>const_buffer u(a);
const_buffer v(a);</pre></td><td><pre>buffer_cast<const void*>(u) == buffer_cast<const void*>(v)
  && buffer_size(u) == buffer_size(v)</pre></td></tr>
<tr><td><pre>const_buffer u(a);
const_buffer v = a;</pre></td><td><pre>buffer_cast<const void*>(u) == buffer_cast<const void*>(v)
  && buffer_size(u) == buffer_size(v)</pre></td></tr>
<tr><td><pre>const_buffer u(a);
const_buffer v; v = a;</pre></td><td><pre>buffer_cast<const void*>(u) == buffer_cast<const void*>(v)
  && buffer_size(u) == buffer_size(v)</pre></td></tr>
<tr><td><pre>const_buffer u(a);</pre></td><td><pre>buffer_cast<const void*>(u) == buffer_cast<const void*>(w)</pre></td></tr>
</table>

expression	postcondition
const X& v = a; const_buffer w(v);	&& buffer_size(u) == buffer_size(w)

const_buffer u(a);
X v(a);
const_buffer w(v);
	buffer_cast<const void*>(u) == buffer_cast<const void*>(w)
	&& buffer_size(u) == buffer_size(w)

const_buffer u(a);
X v = a;
const_buffer w(v);
	buffer_cast<const void*>(u) == buffer_cast<const void*>(w)
	&& buffer_size(u) == buffer_size(w)

const_buffer u(a);
X v(b); v = a;
const_buffer w(v);
	buffer_cast<const void*>(u) == buffer_cast<const void*>(w)
	&& buffer_size(u) == buffer_size(w)

5.5.2.4. Constant buffer sequence requirements

In the table below, X denotes a class containing objects of type T, a denotes a value of type X and u denotes an identifier.

Table--ConstBufferSequence requirements

expression	return type	assertion/note pre/post-condition
X::value_type	T	T meets the requirements for ConvertibleToConstBuffer.
X::const_iterator	iterator type pointing to T	const_iterator meets the requirements for bidirectional iterators (C++ Std, 24.1.4).

X(a);		post: equal_const_buffer_seq(a, X(a)) where the binary predicate equal_const_buffer_seq is defined as
		bool equal_const_buffer_seq(
		  const X& x1, const X& x2)
		{
		  return
		    distance(x1.begin(), x1.end())
		      == distance(x2.begin(), x2.end())
		        && equal(x1.begin(), x1.end(),
		                 x2.begin(), equal_buffer);
		}
		
		and the binary predicate equal_buffer is defined as
		
		bool equal_buffer(
		  const X::value_type& v1,
		  const X::value_type& v2)
		{
		  const_buffer b1(v1);
		  const_buffer b2(v2);
		  return
		    buffer_cast<const void*>(b1)
		      == buffer_cast<const void*>(b2)
		        && buffer_size(b1) == buffer_size(b2);
		}

X u(a);		post:
		distance(a.begin(), a.end())
		  == distance(u.begin(), u.end())
		    && equal(a.begin(), a.end(),
		             u.begin(), equal_buffer)
		
		where the binary predicate equal_buffer is defined as
		
		bool equal_buffer(
		  const X::value_type& v1,
		  const X::value_type& v2)
		{
		  const_buffer b1(v1);
		  const_buffer b2(v2);
		  return
		    buffer_cast<const void*>(b1)
		      == buffer_cast<const void*>(b2)

expression	return type	assertion/note pre/post-condition
		`&& buffer_size(b1) == buffer_size(b2);` `}`
`(&a)->~X();`	`void`	note: the destructor is applied to every element of `a`; all the memory is deallocated.
`a.begin();`	`const_iterator` or convertible to `const_iterator`	
`a.end();`	`const_iterator` or convertible to `const_iterator`	

5.5.2.5. Buffer-oriented synchronous read stream requirements

In the table below, `a` denotes a synchronous read stream object, `mb` denotes an object satisfying mutable buffer sequence requirements, and `ec` denotes an object of type `error_code`.

Table--Buffer-oriented synchronous read stream requirements

operation	type	semantics, pre/post-conditions

`a.read_some(mb);`	`size_t`	Equivalent to:
		`error_code ec;`
		`size_t s = a.read_some(mb, ec);`
		`if (ec) throw system_error(ec);`
		`return s;`

`a.read_some(mb, ec);`	`size_t`	Reads one or more bytes of data from the stream `a`. The mutable buffer sequence `mb` specifies memory where the data should be placed. The `read_some` operation shall always fill a buffer in the sequence completely before proceeding to the next. If successful, returns the number of bytes read and sets `ec` such that `!ec` is true. If an error occurred, returns `0` and sets `ec` such that `!!ec` is true. If the total size of all buffers in the sequence `mb` is `0`, the function shall return `0` immediately.

5.5.2.6. Buffer-oriented asynchronous read stream requirements

In the table below, `a` denotes an asynchronous read stream object, `mb` denotes an object satisfying mutable buffer sequence requirements, and `h` denotes an object satisfying read handler requirements.

Table--Buffer-oriented asynchronous read stream requirements

operation	type	semantics, pre/post-conditions
`a.io_service();`	`io_service&`	Returns the `io_service` object through which the `async_read_some` handler `h` will be invoked.
`a.async_read_some(mb, h);`	`void`	Initiates an asynchronous operation to read one or more bytes of data from the stream `a`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. The mutable buffer sequence `mb` specifies memory where the data should be placed. The `async_read_some` operation shall always fill a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `mb` until such time as the read operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until: — the last copy of `mb` is destroyed, or

operation	type	semantics, pre/post-conditions
		— the handler for the asynchronous read operation is invoked, whichever comes first. If the total size of all buffers in the sequence `mb` is `0`, the asynchronous read operation shall complete immediately and pass `0` as the argument to the handler that specifies the number of bytes read.

5.5.2.7. Buffer-oriented synchronous write stream requirements

In the table below, `a` denotes a synchronous write stream object, `cb` denotes an object satisfying constant buffer sequence requirements, and `ec` denotes an object of type `error_code`.

Table--Buffer-oriented synchronous write stream requirements

operation	type	semantics, pre/post-conditions

`a.write_some(cb);`	`size_t`	Equivalent to:
		`error_code ec;`
		`size_t s = a.write_some(cb, ec);`
		`if (ec) throw system_error(ec);`
		`return s;`

`a.write_some(cb, ec);`	`size_t`	Writes one or more bytes of data to the stream `a`. The constant buffer sequence `cb` specifies memory where the data to be written is located. The `write_some` operation shall always write a buffer in the sequence completely before proceeding to the next. If successful, returns the number of bytes written and sets `ec` such that `!ec` is true. If an error occurred, returns `0` and sets `ec` such that `!!ec` is true. If the total size of all buffers in the sequence `cb` is `0`, the function shall return `0` immediately.

5.5.2.8. Buffer-oriented asynchronous write stream requirements

In the table below, `a` denotes an asynchronous write stream object, `cb` denotes an object satisfying constant buffer sequence requirements, and `h` denotes an object satisfying write handler requirements.

Table--Buffer-oriented asynchronous write stream requirements

operation	type	semantics, pre/post-conditions
`a.io_service();`	`io_service&`	Returns the `io_service` object through which the `async_write_some` handler `h` will be invoked.
`a.async_write_some(cb, h);`	`void`	Initiates an asynchronous operation to write one or more bytes of data to the stream `a`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. The constant buffer sequence `cb` specifies memory where the data to be written is located. The `async_write_some` operation shall always write a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `cb` until such time as the write operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until: — the last copy of `cb` is destroyed, or — the handler for the asynchronous write operation is invoked, whichever comes first.

operation	type	semantics, pre/post-conditions
		If the total size of all buffers in the sequence `cb` is `0`, the asynchronous write operation shall complete immediately and pass `0` as the argument to the handler that specifies the number of bytes written.

5.5.3. Class `mutable_buffer`

The `mutable_buffer` class meets the requirements for `ConvertibleToMutableBuffer` and `ConvertibleToConstBuffer`.

```
namespace std {
  namespace tr2 {
    namespace sys {

      class mutable_buffer
      {
      public:
        mutable_buffer();
        mutable_buffer(void* data, size_t size);
      };

      template<class T> T buffer_cast(const mutable_buffer& b);
      size_t buffer_size(const mutable_buffer& b);

      mutable_buffer operator+(const mutable_buffer& b, size_t size);
      mutable_buffer operator+(size_t size, const mutable_buffer& b);

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.5.3.1. `mutable_buffer` constructors

```
mutable_buffer();
```

Postconditions: `buffer_cast<void*>(*this) == 0` and `buffer_size(*this) == 0`.

```
mutable_buffer(void* data, size_t size);
```

Postconditions: `buffer_cast<void*>(*this) == data` and `buffer_size(*this) == size`.

5.5.3.2. `mutable_buffer` globals

```
template<class T> T buffer_cast(const mutable_buffer& b);
```

Returns: A pointer to the memory area represented by the buffer `b`. `T` must be a pointer type such that the expression `static_cast<T>(p)` is valid, where `p` is of type `void*`.

```
size_t buffer_size(const mutable_buffer& b);
```

Returns: The size of the memory area represented by the buffer `b`.

5.5.3.3. `mutable_buffer` operators

```
mutable_buffer operator+(const mutable_buffer& b, size_t size);
```

Returns: A `mutable_buffer` equivalent to

```
mutable_buffer(
  buffer_cast<char*>(b) + min(size, buffer_size(b)),
  buffer_size(b) - min(size, buffer_size(b)));
```

```
mutable_buffer operator+(size_t size, const mutable_buffer& b);
```

Returns: A `mutable_buffer` equivalent to

```
mutable_buffer(
  buffer_cast<char*>(b) + min(size, buffer_size(b)),
  buffer_size(b) - min(size, buffer_size(b)));
```

5.5.4. Class `const_buffer`

The `const_buffer` class meets the requirements for `ConvertibleToConstBuffer`.

```
namespace std {
  namespace tr2 {
    namespace sys {

      class const_buffer
      {
      public:
        const_buffer();
        const_buffer(const void* data, size_t size);
        const_buffer(const mutable_buffer& b);
      };

      template<class T> T buffer_cast(const const_buffer& b);
      size_t buffer_size(const const_buffer& b);

      const_buffer operator+(const const_buffer& b, size_t size);
      const_buffer operator+(size_t size, const const_buffer& b);

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.5.4.1. `const_buffer` constructors

`const_buffer();`

> *Postconditions:* `buffer_cast<const void*>(*this) == 0` and `buffer_size(*this) == 0`.

`const_buffer(const void* data, size_t size);`

> *Postconditions:* `buffer_cast<const void*>(*this) == data` and `buffer_size(*this) == size`.

`const_buffer(const mutable_buffer& b);`

> *Postconditions:* `buffer_cast<const void*>(*this) == buffer_cast<const void*>(b)` and `buffer_size(*this) == buffer_size(b)`.

5.5.4.2. `const_buffer` globals

`template<class T> T buffer_cast(const const_buffer& b);`

> *Returns:* A pointer to the memory area represented by the buffer `b`. `T` must be a pointer type such that the expression `static_cast<T>(p)` is valid, where `p` is of type `const void*`.

`size_t buffer_size(const const_buffer& b);`

> *Returns:* The size of the memory area represented by the buffer `b`.

5.5.4.3. `const_buffer` operators

`const_buffer operator+(const const_buffer& b, size_t size);`

> *Returns:* A `const_buffer` equivalent to

```
const_buffer(
  buffer_cast<const char*>(b) + min(size, buffer_size(b)),
  buffer_size(b) - min(size, buffer_size(b)));
```

`const_buffer operator+(size_t size, const const_buffer& b);`

> *Returns:* A `const_buffer` equivalent to

```
const_buffer(
  buffer_cast<const char*>(b) + min(size, buffer_size(b)),
  buffer_size(b) - min(size, buffer_size(b)));
```

5.5.5. Class `mutable_buffers_1`

The `mutable_buffers_1` class meets the requirements for `MutableBufferSequence`, `ConstBufferSequence`, `ConvertibleToMutableBuffer`, and `ConvertibleToConstBuffer`.

`mutable_buffers_1` is derived from `mutable_buffer` so that it is convertible to both `mutable_buffer` and `const_buffer`, while ensuring that the conversion to `mutable_buffer` is the better conversion. This prevents ambiguities when passing to functions that are overloaded on both `mutable_buffer` and `const_buffer`, such as `buffer()`.

```
namespace std {
  namespace tr2 {
    namespace sys {

      class mutable_buffers_1 :
        public mutable_buffer
      {
      public:
        // types:
        typedef mutable_buffer value_type;
        typedef unspecified const_iterator;

        // constructors:
        mutable_buffers_1(void* data, size_t size);
        explicit mutable_buffers_1(const mutable_buffer& b);

        // members:
        const_iterator begin() const;
        const_iterator end() const;
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

An object of class `mutable_buffers_1` represents a sequence of exactly one `mutable_buffer` object.

5.5.5.1. `mutable_buffers_1` constructors

```
mutable_buffers_1(const void* data, size_t size);
```

Effects: Constructs an object of class `mutable_buffers_1`, initialising the base class with `mutable_buffer(data, size)`.

```
explicit mutable_buffers_1(const mutable_buffer& b);
```

Effects: Constructs an object of class `mutable_buffers_1`, initialising the base class with `mutable_buffer(b)`.

5.5.5.2. `mutable_buffers_1` members

```
const_iterator begin() const;
```

Returns: An iterator referring to the first (and only) `mutable_buffer` object in the sequence.

```
const_iterator end() const;
```

Returns: An iterator which is the past-the-end value.

5.5.6. Class `const_buffers_1`

The `const_buffers_1` class meets the requirements for `ConstBufferSequence`, and `ConvertibleToConstBuffer`.

```
namespace std {
  namespace tr2 {
    namespace sys {

      class const_buffers_1 :
        public const_buffer
      {
      public:
        // types:
        typedef const_buffer value_type;
        typedef unspecified const_iterator;

        // constructors:
        const_buffers_1(const void* data, size_t size);
        explicit const_buffers_1(const const_buffer& b);

        // members:
        const_iterator begin() const;
        const_iterator end() const;
      };

    } // namespace sys
```

```
  } // namespace tr2
} // namespace std
```

An object of class `const_buffers_1` represents a sequence of exactly one `const_buffer` object.

5.5.6.1. `const_buffers_1` constructors

```
const_buffers_1(const void* data, size_t size);
```

Effects: Constructs an object of class `const_buffers_1`, initialising the base class with `const_buffer(data, size)`.

```
explicit const_buffers_1(const const_buffer& b);
```

Effects: Constructs an object of class `const_buffers_1`, initialising the base class with `const_buffer(b)`.

5.5.6.2. `const_buffers_1` members

```
const_iterator begin() const;
```

Returns: An iterator referring to the first (and only) `const_buffer` object in the sequence.

```
const_iterator end() const;
```

Returns: An iterator which is the past-the-end value.

5.5.7. Buffer creation functions

In the functions below, `T` must be a *POD type*.

For the function overloads below that accept an argument of type `vector<>`, the buffer objects returned are invalidated by any vector operation that also invalidates all references, pointers and iterators referring to the elements in the sequence (C++ Std, 23.2.4).

For the function overloads below that accept an argument of type `basic_string<>`, the buffer objects returned are invalidated according to the rules defined for invalidation of references, pointers and iterators referring to elements of the sequence (C++ Std, 21.3).

```
mutable_buffers_1 buffer(void* p, size_t s);
```

Returns: `mutable_buffers_1(p, s)`.

```
const_buffers_1 buffer(const void* p, size_t s);
```

Returns: `const_buffers_1(p, s)`.

```
mutable_buffers_1 buffer(const mutable_buffer& b);
```

Returns: `mutable_buffers_1(b)`.

```
mutable_buffers_1 buffer(const mutable_buffer& b, size_t s);
```

Returns: A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
  buffer_cast<void*>(b),
  min(buffer_size(b), s));
```

```
const_buffers_1 buffer(const const_buffer& b);
```

Returns: `const_buffers_1(b)`.

```
const_buffers_1 buffer(const const_buffer& b, size_t s);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  buffer_cast<const void*>(b),
  min(buffer_size(b), s));
```

```
template<class T, size_t N>
  mutable_buffers_1 buffer(T (&arr)[N]);
```

Returns: A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
  static_cast<void*>(arr),
  N * sizeof(T));
```

```
template<class T, size_t N>
  mutable_buffers_1 buffer(T (&arr)[N], size_t s);
```

Returns: A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
  static_cast<void*>(arr),
  min(N * sizeof(T), s));
```

```
template<class T, size_t N>
  const_buffers_1 buffer(const T (&arr)[N]);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  static_cast<const void*>(arr),
  N * sizeof(T));
```

```
template<class T, size_t N>
  const_buffers_1 buffer(const T (&arr)[N], size_t s);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  static_cast<const void*>(arr),
  min(N * sizeof(T), s));
```

```
template<class T, size_t N>
  mutable_buffers_1 buffer(array<T, N>& arr);
```

Returns: A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
  arr.data(),
  arr.size() * sizeof(T));
```

```
template<class T, size_t N>
  mutable_buffers_1 buffer(array<T, N>& arr, size_t s);
```

Returns: A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
  arr.data(),
  min(arr.size() * sizeof(T), s));
```

```
template<class T, size_t N>
  const_buffers_1 buffer(array<const T, N>& arr);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  arr.data(),
  arr.size() * sizeof(T));
```

```
template<class T, size_t N>
  const_buffers_1 buffer(array<const T, N>& arr, size_t s);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  arr.data(),
  min(arr.size() * sizeof(T), s));
```

```
template<class T, size_t N>
  const_buffers_1 buffer(const array<T, N>& arr);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  arr.data(),
  arr.size() * sizeof(T));
```

```
template<class T, size_t N>
  const_buffers_1 buffer(const array<T, N>& arr, size_t s);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  arr.data(),
  min(arr.size() * sizeof(T), s));
```

```
template<class T, class Allocator>
  mutable_buffers_1 buffer(vector<T, Allocator>& vec);
```

Returns: A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
  vec.size() ? &vec[0] : 0,
  vec.size() * sizeof(T));
```

```
template<class T, class Allocator>
  mutable_buffers_1 buffer(vector<T, Allocator>& vec, size_t s);
```

Returns: A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
  vec.size() ? &vec[0] : 0,
  min(vec.size() * sizeof(T), s));
```

```
template<class T, class Allocator>
  const_buffers_1 buffer(const vector<T, Allocator>& vec);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  vec.size() ? &vec[0] : 0,
  vec.size() * sizeof(T));
```

```
template<class T, class Allocator>
  const_buffers_1 buffer(const vector<T, Allocator>& vec, size_t s);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  vec.size() ? &vec[0] : 0,
  min(vec.size() * sizeof(T), s));
```

```
template<class CharT, class Traits, class Allocator>
  const_buffers_1 buffer(const basic_string<CharT, Traits, Allocator>& str);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  str.data()
  str.size() * sizeof(CharT));
```

```
template<class CharT, class Traits, class Allocator>
  const_buffers_1 buffer(const basic_string<CharT, Traits, Allocator>& str,
                         size_t s);
```

Returns: A `const_buffers_1` value equivalent to:

```
const_buffers_1(
  str.data()
  min(str.size() * sizeof(CharT), s));
```

5.5.8. Class template `basic_fifobuf`

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Allocator = std::allocator<char> >
      class basic_fifobuf :
        public streambuf
      {
      public:
        // types:
        typedef Allocator allocator_type;
        typedef unspecified const_buffers_type;
        typedef unspecified mutable_buffers_type;
```

```
        // constructors:
        explicit basic_fifobuf(
          size_t max_sz = numeric_limits<size_t>::max(),
          const Allocator& alloc = Allocator());

        // members:
        allocator_type get_allocator() const;
        size_t size() const;
        size_t max_size() const;
        const_buffers_type data() const;
        mutable_buffers_type prepare(size_t n);
        void commit(size_t n);
        void consume(size_t n);

      protected:
        // overridden virtual functions:
        virtual int_type underflow();
        virtual int_type pbackfail(int_type c = traits_type::eof());
        virtual int_type overflow(int_type c = traits_type::eof());

      private:
        basic_fifobuf(const basic_fifobuf&);   // not defined
        void operator=(const basic_fifobuf&);  // not defined
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

The class `basic_fifobuf` is derived from `basic_streambuf` to associate the input sequence and output sequence with one or more objects of some character array type, whose elements store arbitrary values. These character array objects are internal to the `basic_fifobuf` object, but direct access to the array elements is provided to permit them to be used with I/O operations, such as the `send` or `receive` operations of a socket. Characters written to the output sequence of a `basic_fifobuf` object are appended to the input sequence of the same object.

The class `basic_fifobuf` permits the following implementation strategies:

- A single contiguous character array, which is reallocated as necessary to accommodate changes in the size of the character sequence.
- A sequence of one or more character arrays, where each array is of the same size. Additional character array objects are appended to the sequence to accommodate changes in the size of the character sequence.
- A sequence of one or more character arrays of varying sizes. Additional character array objects are appended to the sequence to accommodate changes in the size of the character sequence.

The constructor for `basic_fifobuf` accepts a `size_t` argument specifying the maximum of the sum of the sizes of the input sequence and output sequence. During the lifetime of the `fifobuf` object, the invariant `size() <= max_size()` shall hold. Any member function that would, if successful, cause the invariant to be violated shall throw an exception of class `length_error`.

The constructor for `basic_fifobuf` takes an `Allocator&` argument. A copy of this argument is used for any memory allocation performed, by the constructor and by all member functions, during the lifetime of each `fifobuf` object.

[*Example:* Writing directly from a `fifobuf` to a socket:

```
fifobuf b;
std::ostream os(&b);
os << "Hello, World!\n";

// try sending all data in input sequence
size_t n = sock.send(b.data());

b.consume(n); // sent data is removed from input sequence
```

—*end example*]

[*Example:* Reading from a socket directly into a `fifobuf`:

```
fifobuf b;

// reserve 512 bytes in output sequence
fifobuf::const_buffers_type bufs = b.prepare(512);

size_t n = sock.receive(bufs);

// received data is "committed" from output sequence to input sequence
```

```
b.commit(n);

std::istream is(&b);
std::string s;
is >> s;
```

—*end example*]

5.5.8.1. `basic_fifobuf` constructors

```
explicit basic_fifobuf(
  size_t max_sz = numeric_limits<size_t>::max(),
  const Allocator& alloc = Allocator());
```

Effects: Constructs an object of class `basic_fifobuf<Allocator>`.

Postconditions: The postconditions of this function are indicated in the table below.

Table--basic_fifobuf<Allocator>::basic_fifobuf(size_t, const Allocator&) effects

expression	value
`size()`	`0`
`max_size()`	`max_sz`
`get_allocator()`	`alloc`

5.5.8.2. `basic_fifobuf` members

```
size_t size() const;
```

Returns: The size of the input sequence. The value shall be equal to that calculated for `s` in the following code.

```
size_t s = 0;
const_buffers_type bufs = data();
const_buffers_type::const_iterator i = bufs.begin();
while (i != bufs.end())
{
  const_buffer buf(*i++);
  s += buffer_size(buf);
}
```

Complexity: Constant time. [*Note:* This implies that an implementation based on non-contiguous character array objects must cache the size of all array objects that are part of the input sequence. —*end note*]

```
allocator_type get_allocator() const;
```

Returns: A copy of the `Allocator` object used to construct the `fifobuf`.

```
size_t max_size() const;
```

Returns: The allowed maximum of the sum of the sizes of the input sequence and output sequence.

```
const_buffers_type data() const;
```

Returns: An object of type `const_buffers_type` that satisfies `ConstBufferSequence` requirements, representing all character array objects in the input sequence. The returned object is invalidated by any `fifobuf` member function that modifies the input sequence or output sequence.

```
mutable_buffers_type prepare(size_t n);
```

Requires: `size() + n <= max_size()`.

Effects: Ensures that the output sequence can accommodate `n` characters, reallocating character array objects as necessary.

Returns: An object of type `mutable_buffers_type` that satisfies `MutableBufferSequence` requirements, representing character array objects at the start of the output sequence such that the sum of the buffer sizes is `n`. The returned object is invalidated by any `fifobuf` member function that modifies the input sequence or output sequence.

Throws: `length_error` if `size() + n > max_size()`.

```
void commit(size_t n);
```

Requires: A preceding call `prepare(x)` where `x >= n`, and no intervening operations that modify the input or output sequence.

Effects: Appends `n` characters from the start of the output sequence to the input sequence. The beginning of the output sequence is advanced by `n` characters.

Throws: `length_error` if `n` is greater than the size of the output sequence.

```
void consume(size_t n);
```

Requires: `n <= size()`.

Effects: Removes `n` characters from the beginning of the input sequence.

Throws: `length_error` if `n > size()`.

5.5.8.3. `basic_fifobuf` overridden virtual functions

```
virtual int_type underflow();
```

Effects: Behaves according to the description of `basic_streambuf<char>::underflow()`.

```
virtual int_type pbackfail(int_type c = traits_type::eof());
```

Requires: `size() < max_size()`.

Effects: Behaves according to the description of `basic_streambuf<char>::pbackfail()`, with the specialisation that `length_error` is thrown if prepending the character to the input sequence would require the condition `size() > max_size()` to be true.

```
virtual int_type overflow(int_type c = traits_type::eof());
```

Requires: `traits_type::eq_int_type(c, traits_type::eof()) || size() < max_size()`.

Effects: Behaves according to the description of `basic_streambuf<char>::overflow()`, with the specialisation that `length_error` is thrown if appending the character to the input sequence would require the condition `size() > max_size()` to be true.

5.5.9. Class `transfer_all`

```
namespace std {
  namespace tr2 {
    namespace sys {

      class transfer_all :
        public binary_function<error_code, size_t, bool>
      {
      public:
        bool operator()(const error_code& ec, size_t) const;
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

`operator()` returns `!!ec`.

5.5.10. Class `transfer_at_least`

```
namespace std {
  namespace tr2 {
    namespace sys {

      class transfer_at_least :
        public binary_function<error_code, size_t, bool>
      {
      public:
        transfer_at_least(size_t m);
        bool operator()(const error_code& ec, size_t s) const;
      private:
//      size_t minimum_;  exposition only
      };
```

```
    } // namespace sys
  } // namespace tr2
} // namespace std
```

The constructor initialises `minimum_` with `m`.

`operator()` returns `!!ec || s >= minimum_`.

5.5.11. Synchronous read operations

```
template<class SyncReadStream, class MutableBufferSequence>
  size_t read(SyncReadStream& stream,
              const MutableBufferSequence& buffers);
template<class SyncReadStream, class MutableBufferSequence>
  size_t read(SyncReadStream& stream,
              const MutableBufferSequence& buffers, error_code& ec);
template<class SyncReadStream, class MutableBufferSequence,
  class CompletionCondition>
    size_t read(SyncReadStream& stream,
                const MutableBufferSequence& buffers,
                CompletionCondition completion_condition);
template<class SyncReadStream, class MutableBufferSequence,
  class CompletionCondition>
    size_t read(SyncReadStream& stream,
                const MutableBufferSequence& buffers,
                CompletionCondition completion_condition,
                error_code& ec);
```

Effects: Reads data from the buffer-oriented synchronous read stream object `stream` by performing one or more calls to the stream's `read_some` member function.

The mutable buffer sequence `buffers` specifies memory where the data should be placed. The synchronous read operation shall always fill a buffer in the sequence completely before proceeding to the next.

The `completion_condition` parameter specifies a function object to be called after each call to the stream's `read_some` member function. The function object is passed the `error_code` value from the most recent `read_some` call, and the total number of bytes transferred in the synchronous read operation so far. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

The synchronous read operation continues until:

— all buffers in the mutable buffer sequence `buffers` have been filled; or

— the completion condition returns true.

On exit, `ec` contains the `error_code` value from the most recent `read_some` call.

Returns: The total number of bytes transferred in the synchronous read operation.

```
template<class SyncReadStream, class Allocator>
  size_t read(SyncReadStream& stream, basic_fifobuf<Allocator>& fb);
template<class SyncReadStream, class Allocator>
  size_t read(SyncReadStream& stream, basic_fifobuf<Allocator>& fb,
              error_code& ec);
template<class SyncReadStream, class Allocator,
  class CompletionCondition>
    size_t read(SyncReadStream& stream, basic_fifobuf<Allocator>& fb,
                CompletionCondition completion_condition);
template<class SyncReadStream, class Allocator,
  class CompletionCondition>
    size_t read(SyncReadStream& stream, basic_fifobuf<Allocator>& fb,
                CompletionCondition completion_condition,
                error_code& ec);
```

Effects: Reads data from the synchronous read stream object `stream` by performing one or more calls to the stream's `read_some` member function.

Data is placed into the `basic_fifobuf<>` object `fb`. A mutable buffer sequence is obtained prior to each `read_some` call using `fb.prepare(min(N, fb.max_size() - fb.size()))`, where `N` is a suitable implementation-defined value. After each `read_some` call, the implementation performs `fb.commit(n)`, where `n` is the return value from `read_some`.

The `completion_condition` parameter specifies a function object to be called after each call to the stream's `read_some` member function. The function object is passed the `error_code` value from the most recent `read_some` call, and the total number of bytes transferred in the synchronous read operation so far. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

The synchronous read operation continues until:

— the `basic_fifobuf<>` object `fb` is full, that is `fb.size() == fb.max_size()`; or

— the completion condition returns true.

On exit, `ec` contains the `error_code` value from the most recent `read_some` call.

Returns: The total number of bytes transferred in the synchronous read operation.

5.5.12. Asynchronous read operations

```
template<class AsyncReadStream, class MutableBufferSequence,
  class ReadHandler>
    size_t async_read(AsyncReadStream& stream,
                      const MutableBufferSequence& buffers,
                      ReadHandler handler);
template<class AsyncReadStream, class MutableBufferSequence,
  class CompletionCondition, class ReadHandler>
    size_t async_read(AsyncReadStream& stream,
                      const MutableBufferSequence& buffers,
                      CompletionCondition completion_condition,
                      ReadHandler handler);
```

Effects: Initiates an asynchronous operation to read data from the buffer-oriented asynchronous read stream object `stream` by performing one or more asynchronous operations on the stream using the stream's `async_read_some` member function (henceforth referred to as asynchronous *read_some* operations). The operation is performed via the `io_service` object `stream.io_service()` and behaves according to asynchronous operation requirements.

The mutable buffer sequence `buffers` specifies memory where the data should be placed. The asynchronous read operation shall always fill a buffer in the sequence completely before proceeding to the next.

The implementation shall maintain one or more copies of `buffers` until such time as the read operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until:

— the last copy of `buffers` is destroyed, or

— the handler for the asynchronous operation is invoked,

whichever comes first.

The `completion_condition` parameter specifies a function object to be called after the completion of each asynchronous *read_some* operation. The function object is passed the `error_code` value from the completion handler of the most recent asynchronous *read_some* operation, and the total number of bytes transferred in the asynchronous read operation so far. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

The asynchronous read operation continues until:

— all buffers in the mutable buffer sequence `buffers` have been filled; or

— the completion condition returns true.

The program must ensure the `AsyncReadStream` object `stream` is valid until the handler for the asynchronous operation is invoked.

Any implementation-defined handler objects passed to asynchronous *read_some* operations shall implement `io_handler_allocate`, `io_handler_deallocate` and `io_handler_invoke` such that the calls are forwarded to the equivalent functions for the object `handler`.

On completion of the asynchronous operation, the `ReadHandler` object `handler` is invoked with the `error_code` value from the most recent asynchronous *read_some* operation, and the total number of bytes transferred.

```
template<class AsyncReadStream, class Allocator, class ReadHandler>
  size_t async_read(AsyncReadStream& stream,
                    basic_fifobuf<Allocator>& fb,
                    ReadHandler handler);
template<class AsyncReadStream, class Allocator,
  class CompletionCondition, class ReadHandler>
    size_t async_read(AsyncReadStream& stream,
                      basic_fifobuf<Allocator>& fb,
                      CompletionCondition completion_condition,
                      ReadHandler handler);
```

Effects: Initiates an asynchronous operation to read data from the buffer-oriented asynchronous read stream object `stream` by performing one or more asynchronous *read_some* operations on the stream. The operation is performed via the `io_service` object `stream.io_service()` and behaves according to asynchronous operation requirements.

Data is placed into the `basic_fifobuf<>` object `fb`. A mutable buffer sequence is obtained prior to each `async_read_some` call using `fb.prepare(min(N, fb.max_size() - fb.size()))`, where `N` is a suitable implementation-defined value. After the completion of each asynchronous *read_some* operation, the implementation performs `fb.commit(n)`, where `n` is the value passed to the asynchronous *read_some* operation's completion handler.

The `completion_condition` parameter specifies a function object to be called after the completion of each asynchronous *read_some* operation. The function object is passed the `error_code` value from the completion handler of the most recent asynchronous *read_some* operation, and the total number of bytes transferred in the asynchronous read operation so far. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

The asynchronous read operation continues until:

— the `basic_fifobuf<>` object `fb` is full, that is `fb.size() == fb.max_size()`; or

— the completion condition returns true.

The program must ensure both the `AsyncReadStream` object `stream` and the `basic_fifobuf<>` object `fb` are valid until the handler for the asynchronous operation is invoked.

Any implementation-defined handler objects passed to asynchronous *read_some* operations shall implement `io_handler_allocate`, `io_handler_deallocate` and `io_handler_invoke` such that the calls are forwarded to the equivalent functions for the object `handler`.

On completion of the asynchronous operation, the `ReadHandler` object `handler` is invoked with the `error_code` value from the most recent asynchronous *read_some* operation, and the total number of bytes transferred.

5.5.13. Synchronous write operations

```
template<class SyncWriteStream, class ConstBufferSequence>
  size_t write(SyncWriteStream& stream,
               const ConstBufferSequence& buffers);
template<class SyncWriteStream, class ConstBufferSequence>
  size_t write(SyncWriteStream& stream,
               const ConstBufferSequence& buffers, error_code& ec);
template<class SyncWriteStream, class ConstBufferSequence,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 const ConstBufferSequence& buffers,
                 CompletionCondition completion_condition);
template<class SyncWriteStream, class ConstBufferSequence,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 const ConstBufferSequence& buffers,
                 CompletionCondition completion_condition,
                 error_code& ec);
```

Effects: Writes data to the buffer-oriented synchronous write stream object `stream` by performing one or more calls to the stream's `write_some` member function.

The constant buffer sequence `buffers` specifies memory where the data to be written is located. The synchronous write operation shall always write a buffer in the sequence completely before proceeding to the next.

The `completion_condition` parameter specifies a function object to be called after each call to the stream's `write_some` member function. The function object is passed the `error_code` value from the most recent `write_some` call, and the total number of bytes transferred in the synchronous write operation so far. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

The synchronous write operation continues until:

— all buffers in the constant buffer sequence `buffers` have been written; or

— the completion condition returns true.

On exit, `ec` contains the `error_code` value from the most recent `write_some` call.

Returns: The total number of bytes transferred in the synchronous write operation.

```
template<class SyncWriteStream, class Allocator>
  size_t write(SyncWriteStream& stream, basic_fifobuf<Allocator>& fb);
template<class SyncWriteStream, class Allocator>
  size_t write(SyncWriteStream& stream, basic_fifobuf<Allocator>& fb,
               error_code& ec);
template<class SyncWriteStream, class Allocator,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 basic_fifobuf<Allocator>& fb,
                 CompletionCondition completion_condition);
template<class SyncWriteStream, class Allocator,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 basic_fifobuf<Allocator>& fb,
                 CompletionCondition completion_condition,
                 error_code& ec);
```

Effects: Writes data to the synchronous write stream object `stream` by performing one or more calls to the stream's `write_some` member function.

Data is written from the `basic_fifobuf<>` object `fb`. A constant buffer sequence is obtained prior to each `write_some` call using `fb.data()`. After each `write_some` call, the implementation performs `fb.consume(n)`, where `n` is the return value from `write_some`.

The `completion_condition` parameter specifies a function object to be called after each call to the stream's `write_some` member function. The function object is passed the `error_code` value from the most recent `write_some` call, and the total number of bytes transferred in the synchronous write operation so far. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

The synchronous write operation continues until:

— the `basic_fifobuf<>` object `fb` is empty, that is `fb.size() == 0`; or

— the completion condition returns true.

On exit, `ec` contains the `error_code` value from the most recent `write_some` call.

Returns: The total number of bytes transferred in the synchronous write operation.

5.5.14. Asynchronous write operations

```
template<class AsyncWriteStream, class ConstBufferSequence,
  class WriteHandler>
    size_t async_write(AsyncWriteStream& stream,
                       const ConstBufferSequence& buffers,
                       WriteHandler handler);
template<class AsyncWriteStream, class ConstBufferSequence,
  class CompletionCondition, class WriteHandler>
    size_t async_write(AsyncWriteStream& stream,
                       const ConstBufferSequence& buffers,
                       CompletionCondition completion_condition,
                       WriteHandler handler);
```

Effects: Initiates an asynchronous operation to write data to the buffer-oriented asynchronous write stream object `stream` by performing one or more asynchronous operations on the stream using the stream's `async_write_some` member function (henceforth referred to as asynchronous *write_some* operations). The operation is performed via the `io_service` object `stream.io_service()` and behaves according to asynchronous operation requirements.

The constant buffer sequence `buffers` specifies memory where the data to be written is located. The asynchronous write operation shall always write a buffer in the sequence completely before proceeding to the next.

The implementation shall maintain one or more copies of `buffers` until such time as the write operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until:

— the last copy of `buffers` is destroyed, or

— the handler for the asynchronous operation is invoked,

whichever comes first.

The `completion_condition` parameter specifies a function object to be called after the completion of each asynchronous *write_some* operation. The function object is passed the `error_code` value from the completion handler of the most recent asynchronous *write_some* operation, and the total number of bytes transferred in the asynchronous write operation so far. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

The asynchronous write operation continues until:

— all buffers in the constant buffer sequence `buffers` have been written; or

— the completion condition returns true.

The program must ensure the `AsyncWriteStream` object `stream` is valid until the handler for the asynchronous operation is invoked.

Any implementation-defined handler objects passed to asynchronous *write_some* operations shall implement `io_handler_allocate`, `io_handler_deallocate` and `io_handler_invoke` such that the calls are forwarded to the equivalent functions for the object `handler`.

On completion of the asynchronous operation, the `WriteHandler` object `handler` is invoked with the `error_code` value from the most recent asynchronous *write_some* operation, and the total number of bytes transferred.

```
template<class AsyncWriteStream, class Allocator, class WriteHandler>
  size_t async_write(AsyncWriteStream& stream,
                     basic_fifobuf<Allocator>& fb,
                     WriteHandler handler);
template<class AsyncWriteStream, class Allocator,
  class CompletionCondition, class WriteHandler>
    size_t async_write(AsyncWriteStream& stream,
                       basic_fifobuf<Allocator>& fb,
                       CompletionCondition completion_condition,
                       WriteHandler handler);
```

Effects: Initiates an asynchronous operation to write data to the buffer-oriented asynchronous write stream object `stream` by performing one or more asynchronous *write_some* operations on the stream. The operation is performed via the `io_service` object `stream.io_service()` and behaves according to asynchronous operation requirements.

Data is written from the `basic_fifobuf<>` object `fb`. A constant buffer sequence is obtained prior to each `async_write_some` call using `fb.data()`. After the completion of each asynchronous *write_some* operation, the implementation performs `fb.consume(n)`, where `n` is the value passed to the asynchronous *write_some* operation's completion handler.

The `completion_condition` parameter specifies a function object to be called after the completion of each asynchronous *write_some* operation. The function object is passed the `error_code` value from the completion handler of the most recent asynchronous *write_some* operation, and the total number of bytes transferred in the asynchronous write operation so far. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

The asynchronous write operation continues until:

— the `basic_fifobuf<>` object `fb` is empty, that is `fb.size() == 0`; or

— the completion condition returns true.

The program must ensure both the `AsyncWriteStream` object `stream` and the `basic_fifobuf<>` object `fb` are valid until the handler for the asynchronous operation is invoked.

Any implementation-defined handler objects passed to asynchronous *write_some* operations shall implement `io_handler_allocate`, `io_handler_deallocate` and `io_handler_invoke` such that the calls are forwarded to the equivalent functions for the object `handler`.

On completion of the asynchronous operation, the `WriteHandler` object `handler` is invoked with the `error_code` value from the most recent asynchronous *write_some* operation, and the total number of bytes transferred.

5.5.15. Synchronous delimited read operations

```
template <class SyncReadStream, class Allocator>
  size_t read_until(SyncReadStream& s, basic_fifobuf<Allocator>& fb,
                    char delim);
template <class SyncReadStream, class Allocator>
  size_t read_until(SyncReadStream& s, basic_fifobuf<Allocator>& fb,
                    char delim, error_code& ec);
template <class SyncReadStream, class Allocator>
  size_t read_until(SyncReadStream& s, basic_fifobuf<Allocator>& fb,
                    const string& delim);
template <class SyncReadStream, class Allocator>
  size_t read_until(SyncReadStream& s, basic_fifobuf<Allocator>& fb,
                    const string& delim, error_code& ec);
```

Effects: Reads data from the buffer-oriented synchronous read stream object `stream` by performing zero or more calls to the stream's `read_some` member function, until the `basic_fifobuf<>` object's input sequence contains the specified delimiter `delim`.

Data is placed into the `basic_fifobuf<>` object `fb`. A mutable buffer sequence is obtained prior to each `read_some` call using `fb.prepare(min(N, fb.max_size() - fb.size()))`, where `N` is a suitable implementation-defined value. After each `read_some` call, the implementation performs `fb.commit(n)`, where `n` is the return value from `read_some`.

The synchronous read operation continues until:

— the `basic_fifobuf<>` object's input sequence contains the delimiter `delim`; or

— the `basic_fifobuf<>` object `fb` is full, that is `fb.size() == fb.max_size()`.

On exit, if the `basic_fifobuf<>` object's input sequence contains the delimiter, `ec` shall contain a value such that the expression `!ec` is true. Otherwise, if the `basic_fifobuf<>` object is full, `ec` shall contain the `error_code` value `error::not_found`. If the `basic_fifobuf<>` object is not full, `ec` contains the `error_code` from the most recent `read_some` call.

Returns: The number of bytes in the `basic_fifobuf<>` object's input sequence up to and including the delimiter, if present. Otherwise returns `0`.

5.5.16. Asynchronous delimited read operations

```
template <class AsyncReadStream, class Allocator, class ReadHandler>
  void async_read_until(AsyncReadStream& s,
                        basic_fifobuf<Allocator>& fb, char delim,
                        ReadHandler handler);
template <class AsyncReadStream, class Allocator, class ReadHandler>
  void async_read_until(AsyncReadStream& s,
                        basic_fifobuf<Allocator>& fb,
                        const string& delim, ReadHandler handler);
```

Effects: Initiates an asynchronous operation to read data from the buffer-oriented asynchronous read stream object `stream` by performing zero or more asynchronous *read_some* operations on the stream, until the `basic_fifobuf<>` object's input sequence contains the specified delimiter `delim`. The operation is performed via the `io_service` object `stream.io_service()` and behaves according to asynchronous operation requirements.

Data is placed into the `basic_fifobuf<>` object `fb`. A mutable buffer sequence is obtained prior to each `async_read_some` call using `fb.prepare(min(N, fb.max_size() - fb.size()))`, where `N` is a suitable implementation-defined value. After the completion of each asynchronous *read_some* operation, the implementation performs `fb.commit(n)`, where `n` is the value passed to the asynchronous *read_some* operation's completion handler.

The asynchronous read operation continues until:

— the `basic_fifobuf<>` object's input sequence contains the delimiter `delim`; or

— the `basic_fifobuf<>` object `fb` is full, that is `fb.size() == fb.max_size()`.

The program must ensure both the `AsyncReadStream` object `stream` and the `basic_fifobuf<>` object `fb` are valid until the handler for the asynchronous operation is invoked.

Any implementation-defined handler objects passed to asynchronous *read_some* operations shall implement `io_handler_allocate`, `io_handler_deallocate` and `io_handler_invoke` such that the calls are forwarded to the equivalent functions for the object `handler`.

On completion of the asynchronous operation, the `ReadHandler` object `handler` is invoked with an `error_code` value `ec` and a number of bytes `s`. If the `basic_fifobuf<>` object's input sequence contains the delimiter, `ec` shall contain a value such that the expression `!ec` is true. Otherwise, if the `basic_fifobuf<>` object is full, `ec` shall contain the `error_code` value `error::not_found`. If the `basic_fifobuf<>` object is not full, `ec` contains the `error_code` from the most recent asynchronous *read_some* operation. `s` shall contain the number of bytes in the `basic_fifobuf<>` object's input sequence up to and including the delimiter, if present, otherwise `0`.

5.6. Header `<network>` synopsis

```
namespace std {
  namespace tr2 {
    namespace sys {

      // Sockets:

      class socket_base;

      template<class Protocol, class SocketService>
        class basic_socket;

      template<class Protocol> class datagram_socket_service;

      template<class Protocol,
        class DatagramSocketService = datagram_socket_service<Protocol> >
          class basic_datagram_socket;

      template<class Protocol> class stream_socket_service;

      template<class Protocol,
        class StreamSocketService = stream_socket_service<Protocol> >
          class basic_stream_socket;

      template<class Protocol> class socket_acceptor_service;

      template<class Protocol,
        class SocketAcceptorService = socket_acceptor_service<Protocol> >
          class basic_socket_acceptor;

      // Socket streams:

      template<class Protocol,
        class StreamSocketService = stream_socket_service<Protocol> >
          class basic_socket_streambuf;

      template<class Protocol,
        class StreamSocketService = stream_socket_service<Protocol> >
          class basic_socket_iostream;

      // Internet protocol:
```

```
namespace ip {

  class address;

  // address comparisons:
  bool operator==(const address&, const address&);
  bool operator!=(const address&, const address&);
  bool operator< (const address&, const address&);
  bool operator> (const address&, const address&);
  bool operator<=(const address&, const address&);
  bool operator>=(const address&, const address&);

  // address I/O:
  template<class CharT, class Traits>
    basic_ostream<CharT, Traits>& operator<<(
      basic_ostream<CharT, Traits>&, const address&);

  class address_v4;

  // address_v4 comparisons:
  bool operator==(const address_v4&, const address_v4&);
  bool operator!=(const address_v4&, const address_v4&);
  bool operator< (const address_v4&, const address_v4&);
  bool operator> (const address_v4&, const address_v4&);
  bool operator<=(const address_v4&, const address_v4&);
  bool operator>=(const address_v4&, const address_v4&);

  // address_v4 I/O:
  template<class CharT, class Traits>
    basic_ostream<CharT, Traits>& operator<<(
      basic_ostream<CharT, Traits>&, const address_v4&);

  class address_v6;

  // address_v6 comparisons:
  bool operator==(const address_v6&, const address_v6&);
  bool operator!=(const address_v6&, const address_v6&);
  bool operator< (const address_v6&, const address_v6&);
  bool operator> (const address_v6&, const address_v6&);
  bool operator<=(const address_v6&, const address_v6&);
  bool operator>=(const address_v6&, const address_v6&);

  // address_v6 I/O:
  template<class CharT, class Traits>
    basic_ostream<CharT, Traits>& operator<<(
      basic_ostream<CharT, Traits>&, const address_v6&);

  template<class InternetProtocol>
    class basic_endpoint;

  // basic_endpoint comparisons:
  template<class InternetProtocol>
    bool operator==(const basic_endpoint<InternetProtocol>&,
                    const basic_endpoint<InternetProtocol>&);
  template<class InternetProtocol>
    bool operator!=(const basic_endpoint<InternetProtocol>&,
                    const basic_endpoint<InternetProtocol>&);
  template<class InternetProtocol>
    bool operator< (const basic_endpoint<InternetProtocol>&,
                    const basic_endpoint<InternetProtocol>&);
  template<class InternetProtocol>
    bool operator> (const basic_endpoint<InternetProtocol>&,
                    const basic_endpoint<InternetProtocol>&);
  template<class InternetProtocol>
    bool operator<=(const basic_endpoint<InternetProtocol>&,
                    const basic_endpoint<InternetProtocol>&);
  template<class InternetProtocol>
    bool operator>=(const basic_endpoint<InternetProtocol>&,
                    const basic_endpoint<InternetProtocol>&);

  // basic_endpoint I/O:
  template<class CharT, class Traits, class InternetProtocol>
    basic_ostream<CharT, Traits>& operator<<(
      basic_ostream<CharT, Traits>&,
      const basic_endpoint<InternetProtocol>&);

  class resolver_query_base;
```

```
        template<class InternetProtocol>
          basic_resolver_query;

        template<class InternetProtocol>
          basic_resolver_entry;

        template<class InternetProtocol>
          basic_resolver_iterator;

        template<class InternetProtocol>
          class resolver_service;

        template<class InternetProtocol,
          class ResolverService = resolver_service<InternetProtocol> >
            class basic_resolver;

        string host_name();
        string host_name(error_code&);

        class tcp;

        // tcp comparisons:
        bool operator==(const tcp& a, const tcp& b);
        bool operator!=(const tcp& a, const tcp& b);
        bool operator< (const tcp& a, const tcp& b);
        bool operator> (const tcp& a, const tcp& b);
        bool operator<=(const tcp& a, const tcp& b);
        bool operator>=(const tcp& a, const tcp& b);

        class udp;

        // udp comparisons:
        bool operator==(const udp& a, const udp& b);
        bool operator!=(const udp& a, const udp& b);
        bool operator< (const udp& a, const udp& b);
        bool operator> (const udp& a, const udp& b);
        bool operator<=(const udp& a, const udp& b);
        bool operator>=(const udp& a, const udp& b);

        class v6_only;

        namespace unicast {

          class hops;

        } // namespace unicast

        namespace multicast {

          class join_group;

          class leave_group;

          class outbound_interface;

          class hops;

          class enable_loopback;

        } // namespace multicast
      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.7. Sockets

5.7.1. Requirements

5.7.1.1. Extensibility

This clause defines an optional level of conformance, in the form of additional member functions on types that satisfy `Protocol`, `Endpoint`, `SettableSocketOption`, `GettableSocketOption` or `IoControlCommand` requirements.

[*Note:* When the additional member functions are available, C++ programs may extend the library to add support for other protocols and socket options. —*end note*]

An implementation's level of conformance shall be documented.

[*Note:* Implementations are encouraged to provide the additional member functions, where possible. It is intended that *POSIX* and *Windows* implementations will provide them. —*end note*]

For the purposes of this clause, implementations that provide the additional member functions are known as *extensible implementations*.

5.7.1.2. Endpoint requirements

An endpoint must meet the requirements of `CopyConstructible` types (C++ Std, 20.1.3), and the requirements of `Assignable` types (C++ Std, 23.1).

In the table below, `X` denotes an endpoint class, `a` denotes a value of type `X`, and `u` denotes an identifier.

Table--Endpoint requirements

expression	type	assertion/note pre/post-conditions
`X::protocol_type`	type meeting protocol requirements	
`X u;`		
`X();`		
`a.protocol();`	`protocol_type`	

In the table below, `X` denotes an endpoint class, `a` denotes a value of `X`, `s` denotes a size in bytes, and `u` denotes an identifier.

Table--Endpoint requirements for extensible implementations

expression	type	assertion/note pre/post-conditions
`a.data();`	a pointer	Returns a pointer suitable for passing as the *address* argument to *POSIX* functions such as `accept()`, `getpeername()`, `getsockname()` and `recvfrom()`. The implementation shall perform a `reinterpret_cast` on the pointer to convert it to `sockaddr*`.
`const X& u = a; u.data();`	a pointer	Returns a pointer suitable for passing as the *address* argument to *POSIX* functions such as `connect()`, or as the *dest_addr* argument to *POSIX* functions such as `sendto()`. The implementation shall perform a `reinterpret_cast` on the pointer to convert it to `const sockaddr*`.
`a.size();`	`size_t`	Returns a value suitable for passing as the *address_len* argument to *POSIX* functions such as `connect()`, or as the *dest_len* argument to *POSIX* functions such as `sendto()`, after appropriate integer conversion has been performed.
`a.resize(s);`		post: `a.size() == s` Passed the value contained in the *address_len* argument to *POSIX* functions such as `accept()`, `getpeername()`, `getsockname()` and `recvfrom()`, after successful completion of the function. Permitted to throw an exception if the protocol associated with the endpoint object `a` does not support the specified size.
`a.capacity();`	`size_t`	Returns a value suitable for passing as the *address_len* argument to *POSIX* functions such as `accept()`, `getpeername()`, `getsockname()` and `recvfrom()`, after appropriate integer conversion has been performed.

5.7.1.3. Protocol requirements

A protocol must meet the requirements of `CopyConstructible` types (C++ Std, 20.1.3), and the requirements of `Assignable` types (C++ Std, 23.1).

In the table below, `X` denotes a protocol class, and `a` denotes a value of `X`.

Table--Protocol requirements

expression	return type	assertion/note pre/post-conditions
`X::endpoint`	type meeting endpoint requirements	

In the table below, `X` denotes a protocol class, and `a` denotes a value of `X`.

Table--Protocol requirements for extensible implementations

expression	return type	assertion/note pre/post-conditions
`a.family()`	`int`	Returns a value suitable for passing as the *domain* argument to *POSIX* `socket()` (or equivalent).
`a.type()`	`int`	Returns a value suitable for passing as the *type* argument to *POSIX* `socket()` (or equivalent).
`a.protocol()`	`int`	Returns a value suitable for passing as the *protocol* argument to *POSIX* `socket()` (or equivalent).

5.7.1.4. Socket service requirements

A socket service must meet the requirements for an I/O object service, as well as the additional requirements listed below.

In the table below, `X` denotes a socket service class for protocol `Protocol`, `a` denotes a value of type `X`, `b` denotes a value of type `X::implementation_type`, `p` denotes a value of type `Protocol`, `n` denotes a value of type `X::native_type`, `e` denotes a value of type `Protocol::endpoint`, `ec` denotes a value of type `error_code`, `s` denotes a value meeting `SettableSocketOption` requirements, `g` denotes a value meeting `GettableSocketOption` requirements, `i` denotes a value meeting `IoControlCommand` requirements, `h` denotes a value of type `socket_base::shutdown_type`, `ch` denotes a value meeting `ConnectHandler` requirements, and `u` and `v` denote identifiers.

Table--SocketService requirements

expression	return type	assertion/note pre/post-condition
`X::native_type`		The implementation-defined native representation of a socket. Must satisfy the requirements of `CopyConstructible` types (C++ Std, 20.1.3), and the requirements of `Assignable` types (C++ Std, 23.1).
`a.construct(b);`		From IoObjectService requirements. post: `!a.is_open(b)`.
`a.destroy(b);`		From IoObjectService requirements. Implicitly cancels asynchronous operations, as if by calling `a.close(b, ec)`.
`a.open(b, p, ec);`	`error_code`	pre: `!a.is_open(b)`. post: `!!ec || a.is_open(b)`.
`a.assign(b, p, n, ec);`	`error_code`	pre: `!a.is_open(b)`. post: `!!ec || a.is_open(b)`.
`a.is_open(b);`	`bool`	

`const X& u = a;`
`const X::implementation_type& v = b;`
`u.is_open(v);`
	`bool`	

`a.close(b, ec);`	`error_code`	If `a.is_open()` is true, causes any outstanding asynchronous operations to complete as soon as possible. Handlers for cancelled operations shall be passed the error code `error::operation_aborted`. post: `!a.is_open(b)`.
`a.native(b);`	`X::native_type`	
`a.cancel(b, ec);`	`error_code`	pre: `a.is_open(b)`. Causes any outstanding asynchronous operations to complete as soon as possible.

expression	return type	assertion/note pre/post-condition
		Handlers for cancelled operations shall be passed the error code `error::operation_aborted`.
`a.set_option(b, s, ec);`	`error_code`	pre: `a.is_open(b)`.
`a.get_option(b, g, ec);`	`error_code`	pre: `a.is_open(b)`.

`const X& u = a;`
`const X::implementation_type& v = b;`
`u.get_option(v, g, ec);`
	`error_code`	pre: `a.is_open(b)`.

`a.io_control(b, i, ec);`	`error_code`	pre: `a.is_open(b)`.
`a.at_mark(b, ec);`	`bool`	pre: `a.is_open(b)`.

`const X& u = a;`
`const X::implementation_type& v = b;`
`u.at_mark(v, ec);`
	`bool`	pre: `a.is_open(b)`.

`a.available(b, ec);`	`size_t`	pre: `a.is_open(b)`.

`const X& u = a;`
`const X::implementation_type& v = b;`
`u.available(v, ec);`
	`size_t`	pre: `a.is_open(b)`.

`const typename Protocol::endpoint& u = e;` `a.bind(b, u, ec);`	`error_code`	pre: `a.is_open(b)`.
`a.shutdown(b, h, ec);`	`error_code`	pre: `a.is_open(b)`.
`a.local_endpoint(b, ec);`	`Protocol::endpoint`	pre: `a.is_open(b)`.

`const X& u = a;`
`const X::implementation_type& v = b;`
`u.local_endpoint(v, ec);`
	`Protocol::endpoint`	pre: `a.is_open(b)`.

`a.remote_endpoint(b, ec);`	`Protocol::endpoint`	pre: `a.is_open(b)`.

`const X& u = a;`
`const X::implementation_type& v = b;`
`u.remote_endpoint(v, ec);`
	`Protocol::endpoint`	pre: `a.is_open(b)`.

`const typename Protocol::endpoint& u = e;` `a.connect(b, u, ec);`	`error_code`	pre: `a.is_open(b)`.
`const typename Protocol::endpoint& u = e;` `a.async_connect(b, u, ch);`		pre: `a.is_open(b)`. Initiates an asynchronous connect operation that is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements.

5.7.1.5. Datagram socket service requirements

A datagram socket service must meet the requirements for a socket service, as well as the additional requirements listed below.

In the table below, `X` denotes a datagram socket service class for protocol `Protocol`, `a` denotes a value of type `X`, `b` denotes a value of type `X::implementation_type`, `e` denotes a value of type `Protocol::endpoint`, `ec` denotes a value of type `error_code`, `f` denotes a value of type `socket_base::message_flags`, `mb` denotes a value satisfying mutable buffer sequence requirements, `rh` denotes a value meeting `ReadHandler` requirements, `cb` denotes a value satisfying constant buffer sequence requirements, and `wh` denotes a value meeting `WriteHandler` requirements.

Table--DatagramSocketService requirements

expression	return type	assertion/note pre/post-condition
`a.receive(b, mb, f, ec);`	`size_t`	pre: `a.is_open(b)`. Reads one or more bytes of data from a connected socket `b`. The mutable buffer sequence `mb` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next. If successful, returns the number of bytes read. Otherwise returns `0`.
`a.async_receive(b, mb, f, rh);`	`void`	pre: `a.is_open(b)`. Initiates an asynchronous operation to read one or more bytes of data from a connected socket `b`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. The mutable buffer sequence `mb` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `mb` until such time as the read operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until: — the last copy of `mb` is destroyed, or — the handler for the asynchronous operation is invoked, whichever comes first. If the operation completes successfully, the `ReadHandler` object `rh` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.
`a.receive_from(b, mb, e, f, ec);`	`size_t`	pre: `a.is_open(b)`. Reads one or more bytes of data from an unconnected socket `b`. The mutable buffer sequence `mb` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next. If successful, returns the number of bytes read. Otherwise returns `0`.
`a.async_receive_from(b, mb, e, f, rh);`	`void`	pre: `a.is_open(b)`.

expression	return type	assertion/note pre/post-condition
		Initiates an asynchronous operation to read one or more bytes of data from an unconnected socket `b`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. The mutable buffer sequence `mb` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `mb` until such time as the read operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until: — the last copy of `mb` is destroyed, or — the handler for the asynchronous operation is invoked, whichever comes first. The program must ensure the object `e` is valid until the handler for the asynchronous operation is invoked. If the operation completes successfully, the `ReadHandler` object `rh` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.
`a.send(b, cb, f, ec);`	`size_t`	pre: `a.is_open(b)`. Writes one or more bytes of data to a connected socket `b`. The constant buffer sequence `cb` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next. If successful, returns the number of bytes written. Otherwise returns `0`.
`a.async_send(b, cb, f, wh);`	`void`	pre: `a.is_open(b)`. Initiates an asynchronous operation to write one or more bytes of data to a connected socket `b`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. The constant buffer sequence `cb` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next.

expression	return type	assertion/note pre/post-condition
		The implementation shall maintain one or more copies of `cb` until such time as the write operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until: — the last copy of `cb` is destroyed, or — the handler for the asynchronous operation is invoked, whichever comes first. If the operation completes successfully, the `WriteHandler` object `wh` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.
`const typename Protocol::endpoint& u = e;` `a.send_to(b, cb, u, f, ec);`	`size_t`	pre: `a.is_open(b)`. Writes one or more bytes of data to an unconnected socket `b`. The constant buffer sequence `cb` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next. If successful, returns the number of bytes written. Otherwise returns `0`.
`const typename Protocol::endpoint& u = e;` `a.async_send(b, cb, u, f, wh);`	`void`	pre: `a.is_open(b)`. Initiates an asynchronous operation to write one or more bytes of data to an unconnected socket `b`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. The constant buffer sequence `cb` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `cb` until such time as the write operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until: — the last copy of `cb` is destroyed, or — the handler for the asynchronous operation is invoked, whichever comes first.

expression	return type	assertion/note pre/post-condition
		If the operation completes successfully, the `WriteHandler` object `wh` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.

5.7.1.6. Stream socket service requirements

A stream socket service must meet the requirements for a socket service, as well as the additional requirements listed below.

In the table below, `X` denotes a stream socket service class, `a` denotes a value of type `X`, `b` denotes a value of type `X::implementation_type`, `ec` denotes a value of type `error_code`, `f` denotes a value of type `socket_base::message_flags`, `mb` denotes a value satisfying mutable buffer sequence requirements, `rh` denotes a value meeting `ReadHandler` requirements, `cb` denotes a value satisfying constant buffer sequence requirements, and `wh` denotes a value meeting `WriteHandler` requirements.

Table--StreamSocketService requirements

expression	return type	assertion/note pre/post-condition
`a.receive(b, mb, f, ec);`	`size_t`	pre: `a.is_open(b)`. Reads one or more bytes of data from a connected socket `b`. The mutable buffer sequence `mb` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next. If successful, returns the number of bytes read. Otherwise returns `0`. If the total size of all buffers in the sequence `mb` is `0`, the function shall return `0` immediately. If the operation completes due to graceful connection closure by the peer, the operation shall fail with `error::eof`.
`a.async_receive(b, mb, f, rh);`	`void`	pre: `a.is_open(b)`. Initiates an asynchronous operation to read one or more bytes of data from a connected socket `b`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. The mutable buffer sequence `mb` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `mb` until such time as the read operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until: — the last copy of `mb` is destroyed, or — the handler for the asynchronous operation is invoked, whichever comes first. If the total size of all buffers in the sequence `mb` is `0`, the asynchronous read operation shall complete immediately and pass `0` as the argument to the handler that specifies the number of bytes read. If the operation completes due to graceful connection closure by the peer, the operation shall fail with `error::eof`. If the operation completes successfully, the `ReadHandler` object `rh` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.

expression	return type	assertion/note pre/post-condition
`a.send(b, cb, f, ec);`	`size_t`	pre: `a.is_open(b)`. Writes one or more bytes of data to a connected socket `b`. The constant buffer sequence `cb` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next. If successful, returns the number of bytes written. Otherwise returns `0`. If the total size of all buffers in the sequence `cb` is `0`, the function shall return `0` immediately.
`a.async_send(b, cb, f, wh);`	`void`	pre: `a.is_open(b)`. Initiates an asynchronous operation to write one or more bytes of data to a connected socket `b`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. The constant buffer sequence `cb` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `cb` until such time as the write operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until: — the last copy of `cb` is destroyed, or — the handler for the asynchronous operation is invoked, whichever comes first. If the total size of all buffers in the sequence `cb` is `0`, the asynchronous operation shall complete immediately and pass `0` as the argument to the handler that specifies the number of bytes read. If the operation completes successfully, the `WriteHandler` object `wh` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.

5.7.1.7. Socket acceptor service requirements

A socket acceptor service must meet the requirements for an I/O object service, as well as the additional requirements listed below.

In the table below, `X` denotes a socket acceptor service class for protocol `Protocol`, `a` denotes a value of type `X`, `b` denotes a value of type `X::implementation_type`, `p` denotes a value of type `Protocol`, `n` denotes a value of type `X::native_type`, `e` denotes a value of type `Protocol::endpoint`, `ec` denotes a value of type `error_code`, `s` denotes a value meeting `SettableSocketOption` requirements, `g` denotes a value meeting `GettableSocketOption` requirements, `i` denotes a value meeting `IoControlCommand` requirements, `k` denotes a value of type `basic_socket<Protocol, SocketService>` where `SocketService` is a type meeting socket service requirements, `ah` denotes a value meeting `AcceptHandler` requirements, and `u` and `v` denote identifiers.

Table--SocketAcceptorService requirements

expression	return type	assertion/note pre/post-condition
`X::native_type`		The implementation-defined native representation of a socket acceptor. Must satisfy the requirements of `CopyConstructible` types (C++ Std, 20.1.3), and the requirements of `Assignable` types (C++ Std, 23.1).
`a.construct(b);`		From IoObjectService requirements.

expression	return type	assertion/note pre/post-condition
		post: `!a.is_open(b)`.
`a.destroy(b);`		From IoObjectService requirements. Implicitly cancels asynchronous operations, as if by calling `a.close(b, ec)`.
`a.open(b, p, ec);`	`error_code`	pre: `!a.is_open(b)`. post: `!!ec || a.is_open(b)`.
`a.assign(b, p, n, ec);`	`error_code`	pre: `!a.is_open(b)`. post: `!!ec || a.is_open(b)`.
`a.is_open(b);`	`bool`	

`const X& u = a;`
`const X::implementation_type& v = b;`
`u.is_open(v);`
	`bool`	

`a.close(b, ec);`	`error_code`	If `a.is_open()` is true, causes any outstanding asynchronous operations to complete as soon as possible. Handlers for cancelled operations shall be passed the error code `error::operation_aborted`. post: `!a.is_open(b)`.
`a.native(b);`	`X::native_type`	
`a.cancel(b, ec);`	`error_code`	pre: `a.is_open(b)`. Causes any outstanding asynchronous operations to complete as soon as possible. Handlers for cancelled operations shall be passed the error code `error::operation_aborted`.
`a.set_option(b, s, ec);`	`error_code`	pre: `a.is_open(b)`.
`a.get_option(b, g, ec);`	`error_code`	pre: `a.is_open(b)`.

`const X& u = a;`
`const X::implementation_type& v = b;`
`u.get_option(v, g, ec);`
	`error_code`	pre: `a.is_open(b)`.

`a.io_control(b, i, ec);`	`error_code`	pre: `a.is_open(b)`.
`const typename Protocol::endpoint& u = e;` `a.bind(b, u, ec);`	`error_code`	pre: `a.is_open(b)`.
`a.local_endpoint(b, ec);`	`Protocol::endpoint`	pre: `a.is_open(b)`.

`const X& u = a;`
`const X::implementation_type& v = b;`
`u.local_endpoint(v, ec);`
	`Protocol::endpoint`	pre: `a.is_open(b)`.

`a.accept(b, k, &e, ec);`	`error_code`	pre: `a.is_open(b) && !k.is_open()`. post: `k.is_open()`
`a.accept(b, k, 0, ec);`	`error_code`	pre: `a.is_open(b) && !k.is_open()`. post: `k.is_open()`
`a.async_accept(b, k, &e, ah);`		pre: `a.is_open(b) && !k.is_open()`. Initiates an asynchronous accept operation that is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation

expression	return type	assertion/note pre/post-condition
		requirements. The program must ensure the objects `k` and `e` are valid until the handler for the asynchronous operation is invoked.
`a.async_accept(b, k, 0, ah);`		pre: `a.is_open(b) && !k.is_open()`. Initiates an asynchronous accept operation that is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. The program must ensure the object `k` is valid until the handler for the asynchronous operation is invoked.

5.7.1.8. Gettable socket option requirements

In the table below, `X` denotes a socket option class, `a` denotes a value of `X`, `p` denotes a value that meets the protocol requirements, and `u` denotes an identifier.

Table--GettableSocketOption requirements for extensible implementations

expression	type	assertion/note pre/post-conditions
`a.level(p);`	`int`	Returns a value suitable for passing as the *level* argument to *POSIX* `getsockopt()` (or equivalent).
`a.name(p);`	`int`	Returns a value suitable for passing as the *option_name* argument to *POSIX* `getsockopt()` (or equivalent).
`a.data(p);`	a pointer, convertible to `void*`	Returns a pointer suitable for passing as the *option_value* argument to *POSIX* `getsockopt()` (or equivalent).
`a.size(p);`	`size_t`	Returns a value suitable for passing as the *option_len* argument to *POSIX* `getsockopt()` (or equivalent), after appropriate integer conversion has been performed.
`a.resize(p, s);`		post: `a.size(p) == s`. Passed the value contained in the *option_len* argument to *POSIX* `getsockopt()` (or equivalent) after successful completion of the function. Permitted to throw an exception if the socket option object `a` does not support the specified size.

5.7.1.9. Settable socket option requirements

In the table below, `X` denotes a socket option class, `a` denotes a value of `X`, `p` denotes a value that meets the protocol requirements, and `u` denotes an identifier.

Table--SettableSocketOption requirements for extensible implementations

expression	type	assertion/note pre/post-conditions
`a.level(p);`	`int`	Returns a value suitable for passing as the *level* argument to *POSIX* `setsockopt()` (or equivalent).
`a.name(p);`	`int`	Returns a value suitable for passing as the *option_name* argument to *POSIX* `setsockopt()` (or equivalent).
`const X& u = a;` `u.data(p);`	a pointer, convertible to `const void*`	Returns a pointer suitable for passing as the *option_value* argument to *POSIX* `setsockopt()` (or equivalent).
`a.size(p);`	`size_t`	Returns a value suitable for passing as the *option_len* argument to *POSIX* `setsockopt()` (or equivalent), after appropriate integer conversion has

expression	type	assertion/note pre/post-conditions
		been performed.

5.7.1.10. I/O control command requirements

In the table below, `X` denotes an I/O control command class, `a` denotes a value of `X`, and `u` denotes an identifier.

Table--IoControlCommand requirements for extensible implementations

expression	type	assertion/note pre/post-conditions
`a.name();`	`int`	Returns a value suitable for passing as the *request* argument to *POSIX* `ioctl()` (or equivalent).
`a.data();`	a pointer, convertible to `void*`	

5.7.1.11. Read handler requirements

A read handler must meet the requirements for a handler. A value `h` of a read handler class should work correctly in the expression `h(ec, s)`, where `ec` is an lvalue of type `const error_code` and `s` is an lvalue of type `const size_t`.

5.7.1.12. Write handler requirements

A write handler must meet the requirements for a handler. A value `h` of a write handler class should work correctly in the expression `h(ec, s)`, where `ec` is an lvalue of type `const error_code` and `s` is an lvalue of type `const size_t`.

5.7.1.13. Accept handler requirements

An accept handler must meet the requirements for a handler. A value `h` of an accept handler class should work correctly in the expression `h(ec)`, where `ec` is an lvalue of type `const error_code`.

5.7.1.14. Connect handler requirements

A connect handler must meet the requirements for a handler. A value `h` of a connect handler class should work correctly in the expression `h(ec)`, where `ec` is an lvalue of type `const error_code`.

5.7.2. Class `socket_base`

```
namespace std {
  namespace tr2 {
    namespace sys {

      class socket_base
      {
      public:
        class broadcast;
        class debug;
        class do_not_route;
        class keep_alive;
        class linger;
        class out_of_band_inline;
        class receive_buffer_size;
        class receive_low_watermark;
        class reuse_address;
        class send_buffer_size;
        class send_low_watermark;

        typedef T1 shutdown_type;
        static const shutdown_type shutdown_receive;
        static const shutdown_type shutdown_send;
        static const shutdown_type shutdown_both;

        typedef T2 message_flags;
        static const message_flags message_peek;
        static const message_flags message_out_of_band;
        static const message_flags message_do_not_route;

        static const int max_connections;
```

```
      protected:
        socket_base();
        ~socket_base();
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

`socket_base` defines several member types:

— socket option classes `broadcast`, `debug`, `do_not_route`, `keep_alive`, `linger`, `out_of_band_inline`, `receive_buffer_size`, `receive_low_watermark`, `reuse_address`, `send_buffer_size`, and `send_low_watermark`;

— an enumerated type, `shutdown_type`;

— a bitmask type, `message_flags`.

The value `max_connections` contains the implementation-defined limit on the length of the queue of pending incoming connections.

5.7.3. Boolean socket options

The `socket_base::broadcast`, `socket_base::debug`, `socket_base::do_not_route`, `socket_base::keep_alive`, `socket_base::out_of_band_inline` and `socket_base::reuse_address` classes are boolean socket options.

Boolean socket option classes satisfy the requirements for `CopyConstructible`, `Assignable`, `GettableSocketOption`, and `SettableSocketOption`.

Boolean socket option classes shall be defined as follows:

```
class C
{
public:
  // constructors:
  C();
  explicit C(bool v);

  // members:
  C& operator=(bool v);

  bool value() const;

  operator bool() const;
  bool operator!() const;
};
```

Extensible implementations shall provide the following member functions:

```
class C
{
public:
  template<class Protocol> int level(const Protocol& p) const;
  template<class Protocol> int name(const Protocol& p) const;
  template<class Protocol> unspecified* data(const Protocol& p);
  template<class Protocol> const unspecified* data(const Protocol& p) const;
  template<class Protocol> size_t size(const Protocol& p) const;
  template<class Protocol> void resize(const Protocol& p, size_t s);
  // remainder unchanged
private:
//int value_;  exposition only
};
```

The names and values used in the definition of the boolean socket option classes are described in the table below.

Table--Boolean socket options

C	*L*	*N*	**description**
`socket_base::broadcast`	`SOL_SOCKET`	`SO_BROADCAST`	Determines whether a socket permits sending of broadcast messages, if supported by the protocol.
`socket_base::debug`	`SOL_SOCKET`	`SO_DEBUG`	Determines whether debugging information is recorded by the underlying protocol.

C	*L*	*N*	description
`socket_base::do_not_route`	`SOL_SOCKET`	`SO_DONTROUTE`	Determines whether outgoing messages bypass standard routing facilities.
`socket_base::keep_alive`	`SOL_SOCKET`	`SO_KEEPALIVE`	Determines whether a socket permits sending of keep_alive messages, if supported by the protocol.
`socket_base::out_of_band_inline`	`SOL_SOCKET`	`SO_OOBINLINE`	Determines whether out-of-band data (also known as urgent data) is received inline.
`socket_base::reuse_address`	`SOL_SOCKET`	`SO_REUSEADDR`	Determines whether the validation of endpoints used for binding a socket should allow the reuse of local endpoints, if supported by the protocol.

[*Note:* The constants `SOL_SOCKET`, `SO_BROADCAST`, `SO_DEBUG`, `SO_DONTROUTE`, `SO_KEEPALIVE`, `SO_OOBINLINE` and `SO_REUSEADDR` are defined in the *POSIX* header file `sys/socket.h`. —*end note*]

5.7.3.1. Boolean socket option constructors

`C();`

Postconditions: `!value()`.

`explicit C(bool v);`

Postconditions: `value() == v`.

5.7.3.2. Boolean socket option members

`C& operator=(bool v);`

Returns: `*this`.

Postconditions: `value() == v`.

`bool value() const;`

Returns: The stored socket option value. For extensible implementations, returns `value_ != 0`.

`operator bool() const;`

Returns: `value()`.

`bool operator!() const;`

Returns: `!value()`.

5.7.3.3. Boolean socket option members (extensible implementations)

`template<class Protocol> int level(const Protocol& p) const;`

Returns: ***L***.

`template<class Protocol> int name(const Protocol& p) const;`

Returns: ***N***.

`template<class Protocol> unspecified* data(const Protocol& p);`

Returns: `&value_`.

`template<class Protocol> const unspecified* data(const Protocol& p) const;`

Returns: `&value_`.

`template<class Protocol> size_t size(const Protocol& p) const;`

Returns: `sizeof(value_)`.

`template<class Protocol> void resize(const Protocol& p, size_t s);`

Throws: `length_error` if `s` is not a valid data size for the protocol specified by `p`.

5.7.4. Integral socket options

The `socket_base::receive_buffer_size`, `socket_base::receive_low_watermark`, `socket_base::send_buffer_size` and `socket_base::send_low_watermark` classes are integral socket options.

Integral socket option classes satisfy the requirements for `CopyConstructible`, `Assignable`, `GettableSocketOption`, and `SettableSocketOption`.

Integral socket option classes shall be defined as follows:

```
class C
{
public:
  // constructors:
  C();
  explicit C(int v);

  // members:
  C& operator=(int v);

  int value() const;
};
```

Extensible implementations shall provide the following member functions:

```
class C
{
public:
  template<class Protocol> int level(const Protocol& p) const;
  template<class Protocol> int name(const Protocol& p) const;
  template<class Protocol> unspecified* data(const Protocol& p);
  template<class Protocol> const unspecified* data(const Protocol& p) const;
  template<class Protocol> size_t size(const Protocol& p) const;
  template<class Protocol> void resize(const Protocol& p, size_t s);
  // remainder unchanged
private:
//int value_;   exposition only
};
```

The names and values used in the definition of the integral socket option classes are described in the table below.

Table--Integral socket options

C	*L*	*N*	description
`socket_base::receive_buffer_size`	`SOL_SOCKET`	`SO_RCVBUF`	Specifies the size of the receive buffer associated with a socket.
`socket_base::receive_low_watermark`	`SOL_SOCKET`	`SO_RCVLOWAT`	Specifies the minimum number of bytes to process for socket input operations.
`socket_base::send_buffer_size`	`SOL_SOCKET`	`SO_SNDBUF`	Specifies the size of the send buffer associated with a socket.
`socket_base::send_low_watermark`	`SOL_SOCKET`	`SO_SNDLOWAT`	Specifies the minimum number of bytes to process for socket output operations.

[*Note:* The constants `SOL_SOCKET`, `SO_RCVBUF`, `SO_RCVLOWAT`, `SO_SNDBUF` and `SO_SNDLOWAT` are defined in the *POSIX* header file `sys/socket.h`. *—end note*]

5.7.4.1. Integral socket option constructors

`C();`

Postconditions: `value() == 0`.

`explicit C(int v);`

Postconditions: `value() == v`.

5.7.4.2. Integral socket option members

`C& operator=(int v);`

Returns: `*this`.

Postconditions: `value() == v`.

`int value() const;`

Returns: The stored socket option value. For extensible implementations, returns `value_`.

5.7.4.3. Integral socket option members (extensible implementations)

`template<class Protocol> int level(const Protocol& p) const;`

Returns: ***L***.

`template<class Protocol> int name(const Protocol& p) const;`

Returns: ***N***.

`template<class Protocol> unspecified* data(const Protocol& p);`

Returns: `&value_`.

`template<class Protocol> const unspecified* data(const Protocol& p) const;`

Returns: `&value_`.

`template<class Protocol> size_t size(const Protocol& p) const;`

Returns: `sizeof(value_)`.

`template<class Protocol> void resize(const Protocol& p, size_t s);`

Throws: `length_error` if `s` is not a valid data size for the protocol specified by `p`.

5.7.5. Class `socket_base::linger`

The `linger` class represents a socket option for controlling the behaviour when a socket is closed and unsent data is present.

`linger` satisfies the requirements for `CopyConstructible`, `Assignable`, `GettableSocketOption`, and `SettableSocketOption`.

```
namespace std {
  namespace tr2 {
    namespace sys {

      class socket_base::linger
      {
      public:
        // constructors:
        linger();
        linger(bool e, int t);

        // members:
        bool enabled() const;
        void enabled(bool e);

        int timeout() const;
        void timeout(int t);
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

Extensible implementations shall provide the following member functions:

```
namespace std {
  namespace tr2 {
    namespace sys {

      class socket_base::linger
      {
      public:
        template<class Protocol> int level(const Protocol& p) const;
        template<class Protocol> int name(const Protocol& p) const;
        template<class Protocol> unspecified* data(const Protocol& p);
        template<class Protocol> const unspecified* data(const Protocol& p) const;
```

```
        template<class Protocol> size_t size(const Protocol& p) const;
        template<class Protocol> void resize(const Protocol& p, size_t s);
        // remainder unchanged
      private:
//      ::linger value_;  exposition only
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.7.5.1. `socket_base::linger` constructors

`linger();`

> *Postconditions:* `!enabled() && timeout() == 0`.

`linger(bool e, int t);`

> *Postconditions:* `enabled() == e && timeout() == t`.

5.7.5.2. `socket_base::linger` members

`bool enabled() const;`

> *Returns:* The stored socket option value for whether linger on close is enabled. For extensible implementations, returns `value_.l_onoff != 0`.

`void enabled(bool e);`

> *Postconditions:* `enabled() == e`.

`int timeout() const;`

> *Returns:* The stored socket option value for the linger timeout, in seconds. For extensible implementations, returns `value_.l_linger`.

`void timeout(int t);`

> *Postconditions:* `timeout() == t`.

5.7.5.3. `socket_base::linger` members (extensible implementations)

`template<class Protocol> int level(const Protocol& p) const;`

> *Returns:* `SOL_SOCKET`.
>
> [*Note:* The constant `SOL_SOCKET` is defined in the *POSIX* header file `sys/socket.h`. —*end note*]

`template<class Protocol> int name(const Protocol& p) const;`

> *Returns:* `SO_LINGER`.
>
> [*Note:* The constant `SO_LINGER` is defined in the *POSIX* header file `sys/socket.h`. —*end note*]

`template<class Protocol> unspecified* data(const Protocol& p) const;`

> *Returns:* `&value_`.

`template<class Protocol> const unspecified* data(const Protocol& p) const;`

> *Returns:* `&value_`.

`template<class Protocol> size_t size(const Protocol& p) const;`

> *Returns:* `sizeof(value_)`.

`template<class Protocol> void resize(const Protocol& p, size_t s);`

> *Throws:* `length_error` if `s != sizeof(value_)`.

5.7.6. Class template basic_socket

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Protocol, class SocketService>
      class basic_socket :
        public basic_io_object<SocketService>,
        public socket_base
      {
      public:
        // types:
        typedef typename SocketService::native_type native_type;
        typedef Protocol protocol_type;
        typedef typename Protocol::endpoint endpoint_type;

        // members:
        native_type native();

        void open(const protocol_type& protocol = protocol_type());
        error_code open(const protocol_type& protocol, error_code& ec);

        void assign(const protocol_type& protocol,
                    const native_type& native_socket);
        error_code assign(const protocol_type& protocol,
                          const native_type& native_socket, error_code& ec);

        bool is_open() const;

        void close();
        error_code close(error_code& ec);

        void cancel();
        error_code cancel(error_code& ec);

        template<class SettableSocketOption>
          void set_option(const SettableSocketOption& option);
        template<class SettableSocketOption>
          error_code set_option(const SettableSocketOption& option,
                                error_code& ec);

        template<class GettableSocketOption>
          void get_option(GettableSocketOption& option) const;
        template<class GettableSocketOption>
          error_code get_option(GettableSocketOption& option,
                                error_code& ec) const;

        template<class IoControlCommand>
          void io_control(IoControlCommand& command);
        template<class IoControlCommand>
          error_code io_control(IoControlCommand& command, error_code& ec);

        bool at_mark() const;
        bool at_mark(error_code& ec) const;

        size_t available() const;
        size_t available(error_code& ec) const;

        void bind(const endpoint_type& endpoint);
        error_code bind(const endpoint_type& endpoint, error_code& ec);

        void shutdown(shutdown_type what);
        error_code shutdown(shutdown_type what, error_code& ec);

        endpoint_type local_endpoint() const;
        endpoint_type local_endpoint(error_code& ec) const;

        endpoint_type remote_endpoint() const;
        endpoint_type remote_endpoint(error_code& ec) const;

        void connect(const endpoint_type& endpoint);
        error_code connect(const endpoint_type& endpoint, error_code& ec);

        template<class ConnectHandler>
          void async_connect(const endpoint_type& endpoint,
                             ConnectHandler handler);
```

```
      protected:
        // constructors:
        explicit basic_socket(std::tr2::sys::io_service& io_service);
        basic_socket(std::tr2::sys::io_service& io_service,
                     const protocol_type& protocol);
        basic_socket(std::tr2::sys::io_service& io_service,
                     const endpoint_type& endpoint);
        basic_socket(std::tr2::sys::io_service& io_service,
                     const protocol_type& protocol,
                     const native_type& native_socket);
        ~basic_socket();
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.7.6.1. `basic_socket` constructors

```
explicit basic_socket(std::tr2::sys::io_service& io_service);
```

Effects: Constructs an object of class `basic_socket<Protocol, SocketService>`, initialising the base class with `basic_io_object(io_service)`.

```
basic_socket(std::tr2::sys::io_service& io_service,
             const protocol_type& protocol);
```

Effects: Constructs an object of class `basic_socket<Protocol, SocketService>`, initialising the base class with `basic_io_object(io_service)`, then opening the socket as if by calling:

```
error_code ec;
this->service.open(this->implementation, protocol, ec);
if (ec) throw system_error(ec);
```

```
basic_socket(std::tr2::sys::io_service& io_service,
             const endpoint_type& endpoint);
```

Effects: Constructs an object of class `basic_socket<Protocol, SocketService>`, initialising the base class with `basic_io_object(io_service)`, then opening and binding the socket and marking it as listening as if by calling:

```
error_code ec;
this->service.open(this->implementation, endpoint.protocol(), ec);
if (ec) throw system_error(ec);
this->service.bind(this->implementation, endpoint, ec);
if (ec) throw system_error(ec);
```

```
basic_socket(std::tr2::sys::io_service& io_service,
             const protocol_type& protocol,
             const native_type& native_socket);
```

Effects: Constructs an object of class `basic_socket<Protocol, SocketService>`, initialising the base class with `basic_io_object(io_service)`, then assigning the existing native socket into the object as if by calling:

```
error_code ec;
this->service.assign(this->implementation, protocol, native_socket, ec);
if (ec) throw system_error(ec);
```

5.7.6.2. `basic_socket` members

```
native_type native();
```

Returns: `this->service.native(this->implementation)`.

```
void open(const protocol_type& protocol);
error_code open(const protocol_type& protocol, error_code& ec);
```

Returns: `this->service.open(this->implementation, protocol, ec)`.

```
void assign(const protocol_type& protocol,
            const native_type& native_socket);
error_code assign(const protocol_type& protocol,
                  const native_type& native_socket, error_code& ec);
```

Returns: `this->service.assign(this->implementation, protocol, native_socket, ec)`.

```
bool is_open() const;
```

Returns: `this->service.is_open(this->implementation)`.

```
void close();
error_code close(error_code& ec);
```

Returns: `this->service.close(this->implementation, ec)`.

```
void cancel();
error_code cancel(error_code& ec);
```

Returns: `this->service.cancel(this->implementation, ec)`.

```
template<class SettableSocketOption>
  void set_option(const SettableSocketOption& option);
template<class SettableSocketOption>
  error_code set_option(const SettableSocketOption& option,
                        error_code& ec);
```

Returns: `this->service.set_option(this->implementation, option, ec)`.

```
template<class GettableSocketOption>
  void get_option(GettableSocketOption& option);
template<class GettableSocketOption>
  error_code get_option(GettableSocketOption& option, error_code& ec);
```

Returns: `this->service.get_option(this->implementation, option, ec)`.

```
template<class IoControlCommand>
  void io_control(IoControlCommand& command);
template<class IoControlCommand>
  error_code io_control(IoControlCommand& command, error_code& ec);
```

Returns: `this->service.io_control(this->implementation, command, ec)`.

```
bool at_mark() const;
bool at_mark(error_code& ec) const;
```

Returns: `this->service.at_mark(this->implementation, ec)`.

```
size_t available() const;
size_t available(error_code& ec) const;
```

Returns: `this->service.available(this->implementation, ec)`.

```
void bind(const endpoint_type& endpoint);
error_code bind(const endpoint_type& endpoint, error_code& ec);
```

Returns: `this->service.bind(this->implementation, endpoint, ec)`.

```
void listen(int backlog = max_connections);
error_code listen(int backlog, error_code& ec);
```

Returns: `this->service.listen(this->implementation, backlog, ec)`.

```
void shutdown(shutdown_type what);
error_code shutdown(shutdown_type what, error_code& ec);
```

Returns: `this->service.shutdown(this->implementation, what, ec)`.

```
endpoint_type local_endpoint() const;
endpoint_type local_endpoint(error_code& ec) const;
```

Returns: `this->service.local_endpoint(this->implementation, ec)`.

```
endpoint_type remote_endpoint() const;
endpoint_type remote_endpoint(error_code& ec) const;
```

Returns: `this->service.remote_endpoint(this->implementation, ec)`.

```
void connect(const endpoint_type& endpoint);
error_code connect(const endpoint_type& endpoint, error_code& ec);
```

Effects: If `is_open()` is false, opens the socket by calling:

```
this->service.open(this->implementation, endpoint.protocol(), ec);
```

then, if no error has occurred, connects the socket by calling:

```
this->service.connect(this->implementation, endpoint, ec);
```

Returns: `ec`.

```
template<class ConnectHandler>
  void async_connect(const endpoint_type& endpoint, ConnectHandler handler);
```

Effects: If `is_open()` is false, opens the socket by calling:

```
error_code ec;
this->service.open(this->implementation, endpoint.protocol(), ec);
```

If the socket failed to open, uses `this->service.io_service().post(...)` to invoke the handler with the `error_code` value `ec`. Otherwise initiates an asynchronous operation to connect the socket by calling:

```
this->service.async_connect(this->implementation, endpoint, handler);
```

5.7.7. Class template `datagram_socket_service`

Instances of the `datagram_socket_service` class template meet the requirements of a DatagramSocketService.

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Protocol>
      class datagram_socket_service :
        public io_service::service
      {
      public:
        static io_service::id id;

        // types:
        typedef Protocol protocol_type;
        typedef typename Protocol::endpoint endpoint_type;
        typedef unspecified implementation_type;
        typedef implementation defined native_type;

        // constructors:
        explicit datagram_socket_service(io_service& ios);

        // members:
        void construct(implementation_type& impl);

        void destroy(implementation_type& impl);

        error_code open(implementation_type& impl,
                        const protocol_type& protocol, error_code& ec);

        error_code assign(implementation_type& impl,
                          const protocol_type& protocol,
                          const native_type& native_socket, error_code& ec);

        bool is_open(const implementation_type& impl) const;

        error_code close(implementation_type& impl, error_code& ec);

        native_type native(implementation_type& impl);

        error_code cancel(implementation_type& impl, error_code& ec);

        template<class SettableSocketOption>
          error_code set_option(implementation_type& impl,
                                const SettableSocketOption& option,
                                error_code& ec);

        template<class GettableSocketOption>
          error_code get_option(const implementation_type& impl,
                                GettableSocketOption& option,
                                error_code& ec) const;

        template<class IoControlCommand>
          error_code io_control(implementation_type& impl,
                                IoControlCommand& command, error_code& ec);
```

```
  bool at_mark(const implementation_type& impl, error_code& ec) const;

  size_t available(const implementation_type& impl,
                   error_code& ec) const;

  error_code bind(implementation_type& impl,
                  const endpoint_type& endpoint, error_code& ec);

  error_code shutdown(implementation_type& impl, shutdown_type how,
                      error_code& ec);

  endpoint_type local_endpoint(const implementation_type& impl,
                               error_code& ec) const;

  endpoint_type remote_endpoint(const implementation_type& impl,
                                error_code& ec) const;

  error_code connect(implementation_type& impl,
                     const endpoint_type& endpoint, error_code& ec);

  template<class ConnectHandler>
    void async_connect(implementation_type& impl,
                       const endpoint_type& endpoint,
                       ConnectHandler handler);

  template<class MutableBufferSequence>
    size_t receive(implementation_type& impl,
                   const MutableBufferSequence& buffers,
                   socket_base::message_flags flags,
                   error_code& ec);

  template<class MutableBufferSequence, class ReadHandler>
    void async_receive(implementation_type& impl,
                       const MutableBufferSequence& buffers,
                       socket_base::message_flags flags,
                       ReadHandler handler);

  template<class MutableBufferSequence>
    size_t receive_from(implementation_type& impl,
                        const MutableBufferSequence& buffers,
                        endpoint_type& sender,
                        socket_base::message_flags flags,
                        error_code& ec);

  template<class MutableBufferSequence, class ReadHandler>
    void async_receive_from(implementation_type& impl,
                            const MutableBufferSequence& buffers,
                            endpoint_type& sender,
                            socket_base::message_flags flags,
                            ReadHandler handler);

  template<class ConstBufferSequence>
    size_t send(implementation_type& impl,
                const ConstBufferSequence& buffers,
                socket_base::message_flags flags,
                error_code& ec);

  template<class ConstBufferSequence, class WriteHandler>
    void async_send(implementation_type& impl,
                    const ConstBufferSequence& buffers,
                    socket_base::message_flags flags,
                    WriteHandler handler);

  template<class ConstBufferSequence>
    size_t send_to(implementation_type& impl,
                   const ConstBufferSequence& buffers,
                   const endpoint_type& destination,
                   socket_base::message_flags flags,
                   error_code& ec);

  template<class ConstBufferSequence, class WriteHandler>
    void async_send_to(implementation_type& impl,
                       const ConstBufferSequence& buffers,
                       const endpoint_type& destination,
                       socket_base::message_flags flags,
                       WriteHandler handler);

private:
```

```
        virtual void shutdown_service();
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.7.7.1. `datagram_socket_service` types

```
typedef implementation defined native_type;
```

The native representation of a socket. Must satisfy the requirements of `CopyConstructible` types (C++ Std, 20.1.3), and the requirements of `Assignable` types (C++ Std, 23.1). [*Note:* For *POSIX* implementations, this type would typically be convertible to and from `int`. For *Windows* implementations, this type would typically be convertible to and from `SOCKET`. *—end note*]

5.7.7.2. `datagram_socket_service` constructors

```
explicit datagram_socket_service(io_service& ios);
```

Effects: Constructs an object of class `datagram_socket_service<Protocol>`, initialising the base class with `io_service::service(ios)`.

5.7.7.3. `datagram_socket_service` members

```
void shutdown_service();
```

Effects: Destroys all copies of user-defined handler objects owned by the service.

```
void construct(implementation_type& impl);
```

Effects: Initialises the socket implementation `impl`.

Postconditions: `!is_open(impl)`.

```
void destroy(implementation_type& impl);
```

Effects: If `is_open(impl)` is true, cancels pending asynchronous operations associated with `impl`, disables the linger socket option to prevent the operation from blocking, and releases socket resources as if by *POSIX* `close()`. Otherwise, no effect. Handlers for cancelled asynchronous operations are passed the `error_code` value `error::operation_aborted`. Failures are ignored.

```
error_code open(implementation_type& impl,
                const protocol_type& protocol, error_code& ec);
```

Requires: `!is_open(impl)`.

Effects: If `is_open(impl)` is true, sets `ec` to `error::already_open`. Otherwise establishes the postcondition, as if by *POSIX* `socket()`.

Returns: `ec`.

Postconditions: `is_open(impl)`.

```
error_code assign(implementation_type& impl,
                  const protocol_type& protocol,
                  const native_type& native_socket, error_code& ec);
```

Requires: `!is_open(impl)`.

Effects: If `is_open(impl)` is true, sets `ec` to `error::already_open`. Otherwise assigns the existing socket to the implementation `impl`.

Returns: `ec`.

Postconditions: `is_open(impl)`.

The main source of errors for `assign` would be a call to register the socket with an OS-specific event demultiplexor, such as a `kqueue`, an `epoll` descriptor, a `/dev/poll` device, or a Windows I/O completion port. These errors may also be produced by `open`, since that function would perform the same registration.

```
bool is_open(const implementation_type& impl) const;
```

Returns: A `bool` indicating whether the socket implementation `impl` was opened by a previous call to `open` or `assign`.

```
error_code close(implementation_type& impl, error_code& ec);
```

Effects: If `is_open(impl)` is true, cancels pending asynchronous operations associated with `impl`, and establishes the postcondition as if by *POSIX* `close()`. Otherwise, no effect. Handlers for cancelled asynchronous operations are passed the `error_code` value `error::operation_aborted`.

Returns: `ec`.

```
native_type native(implementation_type& impl);
```

Returns: The native representation of the socket implementation `impl`.

```
error_code cancel(implementation_type& impl, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise cancels pending asynchronous operations associated with `impl`, if any. Handlers for cancelled asynchronous operations are passed the `error_code` value `error::operation_aborted`.

The conditions under which cancellation of asynchronous operations is permitted are implementation-defined. If current conditions do not permit cancellation, an implementation shall set `ec` to `error::operation_not_supported`.

This flexibility is included to support implementations on Windows versions prior to Vista, where the `CancelIo` function will only cancel asynchronous operations started from the same thread. Vista provides `CancelIoEx` which may be used to cancel all asynchronous operations associated with a socket.

Returns: `ec`.

```
template<class SettableSocketOption>
  error_code set_option(implementation_type& impl,
                        const SettableSocketOption& option,
                        error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise sets an option on the socket `impl`, as if by *POSIX* `setsockopt()`.

Returns: `ec`.

```
template<class GettableSocketOption>
  error_code get_option(const implementation_type& impl,
                        GettableSocketOption& option,
                        error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise gets an option from the socket `impl`, as if by *POSIX* `getsockopt()`.

Returns: `ec`.

```
template<class IoControlCommand>
  error_code io_control(implementation_type& impl,
                        IoControlCommand& command, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise executes an I/O control command on the socket `impl`, as if by *POSIX* `ioctl()`.

Returns: `ec`.

This proposal does not include any classes that satisfy `IoControlCommand` requirements. However, implementation-specific extensions such as QoS may be implemented using `ioctl()`, and the `io_control()` operation is included to allow these extensions to be supported.

```
bool at_mark(const implementation_type& impl, error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. If the value returned by `native(impl)` is not a valid socket, sets `ec` to `error::not_socket`. [*Note:* Instead of *POSIX* `ENOTTY`. *—end note*] Otherwise determines if the socket `impl` is at the out-of-band data mark, as if by *POSIX* `sockatmark()`.

Returns: A `bool` indicating whether the socket is at the out-of-band data mark.

```
size_t available(const implementation_type& impl,
                 error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise determines the number of bytes that may be read without blocking.

Returns: The number of bytes that may be read without blocking, or 0 if an error occurs.

```
error_code bind(implementation_type& impl,
                const endpoint_type& endpoint, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise binds the socket `impl` to the specified local endpoint, as if by *POSIX* `bind()`.

Returns: `ec`.

```
error_code shutdown(implementation_type& impl, shutdown_type how,
                    error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise shuts down all or part of a full-duplex connection for the socket `impl`, as if by *POSIX* `shutdown()`. If `how` is `shutdown_receive`, uses *POSIX* `SHUT_RD`. If `how` is `shutdown_send`, uses *POSIX* `SHUT_WR`. If `how` is `shutdown_both`, uses *POSIX* `SHUT_RDWR`.

Returns: `ec`.

```
endpoint_type local_endpoint(const implementation_type& impl,
                             error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise determines the locally-bound endpoint associated with the socket `impl`, as if by *POSIX* `getsockname()`.

Returns: On success, the locally-bound endpoint associated with `impl`. Otherwise `endpoint_type()`.

```
endpoint_type remote_endpoint(const implementation_type& impl,
                              error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise determines the remote endpoint associated with the socket `impl`, as if by *POSIX* `getpeername()`.

Returns: On success, the remote endpoint associated with `impl`. Otherwise `endpoint_type()`.

```
error_code connect(implementation_type& impl,
                   const endpoint_type& endpoint, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise connects the socket `impl` to the specified remote endpoint, as if by *POSIX* `connect()`.

Returns: `ec`.

```
template<class ConnectHandler>
  void async_connect(implementation_type& impl,
                     const endpoint_type& endpoint,
                     ConnectHandler handler);
```

Requires: `is_open(impl)`.

Effects: Initiates an asynchronous operation to connect the socket `impl` to the specified remote endpoint, as if by *POSIX* `connect()`. The operation is performed via the `io_service` object returned by `io_service::service::io_service()` and behaves according to asynchronous operation requirements.

If `is_open(impl)` is false, the operation shall fail immediately with `error_code` value `error::bad_descriptor`.

When multiple asynchronous connect operations are initiated such that the operations may logically be performed in parallel, the behaviour is implementation-defined. When an asynchronous connect operation and an asynchronous read or write operation are initiated such that they may logically be performed in parallel, the behaviour is implementation-defined.

If a program performs a synchronous `set_option`, `io_control`, `bind`, `shutdown`, `connect`, `receive`, `receive_from`, `send` or `send_to` operation on `impl` while there is an incomplete asynchronous connect operation, the behaviour is implementation-defined.

```
template<class MutableBufferSequence>
  size_t receive(implementation_type& impl,
                 const MutableBufferSequence& buffers,
                 socket_base::message_flags flags,
                 error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise reads data from a connected socket `impl`, as if by *POSIX* `recvmsg()`.

The mutable buffer sequence `buffers` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.

The `flags` argument specifies the type of receive operation to be performed.

If the operation completes due to graceful connection closure by the peer, the operation shall fail with `error::eof`.

Returns: On success, the number of bytes read. Otherwise `0`.

```
template<class MutableBufferSequence, class ReadHandler>
  void async_receive(implementation_type& impl,
                     const MutableBufferSequence& buffers,
                     socket_base::message_flags flags,
                     ReadHandler handler);
```

Requires: `is_open(impl) && (flags & socket_base::message_peek) == 0`.

Effects: Initiates an asynchronous operation to read data from a connected socket `impl`, as if by *POSIX* `recvmsg()`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements.

If `is_open(impl)` is false, the operation shall fail immediately with `error_code` value `error::bad_descriptor`.

The mutable buffer sequence `buffers` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `buffers` until such time as the read operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until:

— the last copy of `buffers` is destroyed, or

— the handler for the asynchronous read operation is invoked,

whichever comes first.

The `flags` argument specifies the type of receive operation to be performed. If `flags &`

`socket_base::message_peek` is true, the operation shall fail immediately with `error::invalid_argument`. If `flags & socket_base::message_out_of_band` is true, the operation shall continue until out-of-band data is received, or an error occurs.

When multiple asynchronous read operations with zero `flags` are initiated such that the operations may logically be performed in parallel, the implementation shall fill the `buffers` in the order in which the operations were issued. The order of invocation of the handlers for these operations is implementation-defined. When multiple asynchronous read operations with non-zero `flags` are initiated such that operations may logically be performed in parallel, the behaviour is implementation-defined.

If a program performs a synchronous `set_option`, `io_control`, `bind`, `shutdown`, `connect`, `receive`, or `receive_from` operation on `impl` while there is an incomplete asynchronous read operation, the behaviour is implementation-defined.

If the operation completes due to graceful connection closure by the peer, the operation shall fail with `error::eof`.

If the operation completes successfully, the `ReadHandler` object `handler` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.

```
template<class MutableBufferSequence>
  size_t receive_from(implementation_type& impl,
                      const MutableBufferSequence& buffers,
                      endpoint_type& sender,
                      socket_base::message_flags flags,
                      error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise reads data from an unconnected socket `impl`, as if by *POSIX* `recvmsg()`.

The mutable buffer sequence `buffers` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.

The `flags` argument specifies the type of receive operation to be performed.

Returns: On success, the number of bytes read. Otherwise `0`.

```
template<class MutableBufferSequence, class ReadHandler>
  void async_receive_from(implementation_type& impl,
                          const MutableBufferSequence& buffers,
                          endpoint_type& sender,
                          socket_base::message_flags flags,
                          ReadHandler handler);
```

Requires: `is_open(impl) && (flags & socket_base::message_peek) == 0`.

Effects: Initiates an asynchronous operation to read data from an unconnected socket `impl`, as if by *POSIX* `recvmsg()`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements.

If `is_open(impl)` is false, the operation shall fail immediately with `error_code` value `error::bad_descriptor`.

The mutable buffer sequence `buffers` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `buffers` until such time as the read operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until:

— the last copy of `buffers` is destroyed, or

— the handler for the asynchronous read operation is invoked,

whichever comes first.

The `flags` argument specifies the type of receive operation to be performed. If `flags & socket_base::message_peek` is true, the operation shall fail immediately with `error::invalid_argument`. If `flags & socket_base::message_out_of_band` is true, the operation shall continue until out-of-band data is received, or an error occurs.

When multiple asynchronous read operations with zero `flags` are initiated such that the operations may logically be performed in parallel, the implementation shall fill the `buffers` in the order in which the operations were issued. The order of invocation of the handlers for these operations is implementation-defined. When multiple asynchronous read operations with non-zero `flags` are initiated such that operations may logically be performed in parallel, the behaviour is implementation-defined.

If a program performs a synchronous `set_option`, `io_control`, `bind`, `shutdown`, `connect`, `receive`, or `receive_from` operation on `impl` while there is an incomplete asynchronous read operation, the behaviour is implementation-defined.

The program must ensure the object `sender` is valid until the handler for the asynchronous operation is invoked.

If the operation completes successfully, the `ReadHandler` object `handler` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.

```
template<class ConstBufferSequence>
  size_t send(implementation_type& impl,
              const ConstBufferSequence& buffers,
              socket_base::message_flags flags,
              error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise writes data to a connected socket `impl`, as if by *POSIX* [`sendmsg()`].

The constant buffer sequence `buffers` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next.

The `flags` argument specifies the type of send operation to be performed.

Returns: On success, the number of bytes written. Otherwise `0`.

```
template<class ConstBufferSequence, class WriteHandler>
  void async_send(implementation_type& impl,
                  const ConstBufferSequence& buffers,
                  socket_base::message_flags flags,
                  WriteHandler handler);
```

Requires: `is_open(impl)`.

Effects: Initiates an asynchronous operation to write data to a connected socket `impl`, as if by *POSIX* [`sendmsg()`]. The operation is performed via the `io_service` object `a.io_service()` and behaves according to [asynchronous operation] requirements.

If `is_open(impl)` is false, the operation shall fail immediately with `error_code` value `error::bad_descriptor`.

The constant buffer sequence `buffers` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `buffers` until such time as the write operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until:

— the last copy of `buffers` is destroyed, or

— the handler for the asynchronous write operation is invoked,

whichever comes first.

The `flags` argument specifies the type of send operation to be performed.

When multiple asynchronous write operations with zero `flags` are initiated such that the operations may logically be performed in parallel, the implementation shall write the `buffers` in the order in which the operations were issued. The order of invocation of the handlers for these operations is implementation-defined. When multiple asynchronous write operations with non-zero `flags` are initiated such that operations may logically be performed in parallel, the behaviour is implementation-defined.

If a program performs a synchronous `set_option`, `io_control`, `bind`, `shutdown`, `connect`, `send`, or `send_to` operation on `impl` while there is an incomplete asynchronous write operation, the behaviour is implementation-defined.

If the operation completes successfully, the `WriteHandler` object `handler` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.

```
template<class ConstBufferSequence>
  size_t send_to(implementation_type& impl,
                 const ConstBufferSequence& buffers,
                 const endpoint_type& destination,
                 socket_base::message_flags flags,
                 error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise writes data to an unconnected socket `impl`, as if by *POSIX* `sendmsg()`.

The constant buffer sequence `buffers` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next.

The `flags` argument specifies the type of send operation to be performed.

Returns: On success, the number of bytes written. Otherwise `0`.

```
template<class ConstBufferSequence, class WriteHandler>
  void async_send_to(implementation_type& impl,
                     const ConstBufferSequence& buffers,
                     const endpoint_type& destination,
                     socket_base::message_flags flags,
                     WriteHandler handler);
```

Requires: `is_open(impl)`.

Effects: Initiates an asynchronous operation to write data to an unconnected socket `impl`, as if by *POSIX* `sendmsg()`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements.

If `is_open(impl)` is false, the operation shall fail immediately with `error_code` value `error::bad_descriptor`.

The constant buffer sequence `buffers` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `buffers` until such time as the write operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until:

— the last copy of `buffers` is destroyed, or

— the handler for the asynchronous write operation is invoked,

whichever comes first.

The `flags` argument specifies the type of send operation to be performed.

When multiple asynchronous write operations with zero `flags` are initiated such that the operations may logically be performed in parallel, the implementation shall write the `buffers` in the order in which the operations were issued. The order of invocation of the handlers for these operations is implementation-defined. When multiple asynchronous write operations with non-zero `flags` are initiated such that operations may logically be performed in parallel, the behaviour is implementation-defined.

If a program performs a synchronous `set_option`, `io_control`, `bind`, `shutdown`, `connect`, `send`, or `send_to` operation on `impl` while there is an incomplete asynchronous write operation, the behaviour is implementation-defined.

If the operation completes successfully, the `WriteHandler` object `handler` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.

5.7.8. Class template `basic_datagram_socket`

```
namespace std {
  namespace tr2 {
    namespace sys {
```

```
template<class Protocol, class DatagramSocketService>
class basic_datagram_socket :
  public basic_socket<DatagramSocketService>
{
public:
  // types:
  typedef typename DatagramSocketService::native_type native_type;
  typedef Protocol protocol_type;
  typedef typename Protocol::endpoint endpoint_type;

  // constructors:
  explicit basic_datagram_socket(std::tr2::sys::io_service& io_service);
  basic_datagram_socket(std::tr2::sys::io_service& io_service,
                        const protocol_type& protocol);
  basic_datagram_socket(std::tr2::sys::io_service& io_service,
                        const endpoint_type& endpoint);
  basic_datagram_socket(std::tr2::sys::io_service& io_service,
                        const protocol_type& protocol,
                        const native_type& native_socket);

  // members:
  template<class MutableBufferSequence>
    size_t receive(const MutableBufferSequence& buffers);
  template<class MutableBufferSequence>
    size_t receive(const MutableBufferSequence& buffers,
                   error_code& ec);

  template<class MutableBufferSequence, class ReadHandler>
    void async_receive(const MutableBufferSequence& buffers,
                       ReadHandler handler);

  template<class MutableBufferSequence>
    size_t receive(const MutableBufferSequence& buffers,
                   socket_base::message_flags flags);
  template<class MutableBufferSequence>
    size_t receive(const MutableBufferSequence& buffers,
                   socket_base::message_flags flags, error_code& ec);

  template<class MutableBufferSequence, class ReadHandler>
    void async_receive(const MutableBufferSequence& buffers,
                       socket_base::message_flags flags,
                       ReadHandler handler);

  template<class MutableBufferSequence>
    size_t receive_from(const MutableBufferSequence& buffers,
                        endpoint_type& sender);
  template<class MutableBufferSequence>
    size_t receive_from(const MutableBufferSequence& buffers,
                        endpoint_type& sender, error_code& ec);

  template<class MutableBufferSequence, class ReadHandler>
    void async_receive_from(const MutableBufferSequence& buffers,
                            endpoint_type& sender,
                            ReadHandler handler);

  template<class MutableBufferSequence>
    size_t receive_from(const MutableBufferSequence& buffers,
                        endpoint_type& sender,
                        socket_base::message_flags flags);
  template<class MutableBufferSequence>
    size_t receive_from(const MutableBufferSequence& buffers,
                        endpoint_type& sender,
                        socket_base::message_flags flags,
                        error_code& ec);

  template<class MutableBufferSequence, class ReadHandler>
    void async_receive_from(const MutableBufferSequence& buffers,
                            endpoint_type& sender,
                            socket_base::message_flags flags,
                            ReadHandler handler);

  template<class ConstBufferSequence>
    size_t send(const ConstBufferSequence& buffers);
  template<class ConstBufferSequence>
    size_t send(const ConstBufferSequence& buffers, error_code& ec);

  template<class ConstBufferSequence, class WriteHandler>
    void async_send(const ConstBufferSequence& buffers,
```

```
                          WriteHandler handler);

        template<class ConstBufferSequence>
          size_t send(const ConstBufferSequence& buffers,
                      socket_base::message_flags flags);
        template<class ConstBufferSequence>
          size_t send(const ConstBufferSequence& buffers,
                      socket_base::message_flags flags, error_code& ec);

        template<class ConstBufferSequence, class WriteHandler>
          void async_send(const ConstBufferSequence& buffers,
                          socket_base::message_flags flags,
                          WriteHandler handler);

        template<class ConstBufferSequence>
          size_t send_to(const ConstBufferSequence& buffers,
                         const endpoint_type& destination);
        template<class ConstBufferSequence>
          size_t send_to(const ConstBufferSequence& buffers,
                         const endpoint_type& destination, error_code& ec);

        template<class ConstBufferSequence, class WriteHandler>
          void async_send_to(const ConstBufferSequence& buffers,
                             const endpoint_type& destination,
                             WriteHandler handler);

        template<class ConstBufferSequence>
          size_t send_to(const ConstBufferSequence& buffers,
                         const endpoint_type& destination,
                         socket_base::message_flags flags);
        template<class ConstBufferSequence>
          size_t send_to(const ConstBufferSequence& buffers,
                         const endpoint_type& destination,
                         socket_base::message_flags flags, error_code& ec);

        template<class ConstBufferSequence, class WriteHandler>
          void async_send_to(const ConstBufferSequence& buffers,
                             const endpoint_type& destination,
                             socket_base::message_flags flags,
                             WriteHandler handler);
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.7.8.1. `basic_datagram_socket` constructors

```
explicit basic_datagram_socket(std::tr2::sys::io_service& io_service);
```

Effects: Constructs an object of class `basic_datagram_socket<Protocol, DatagramSocketService>`, initialising the base class with `basic_socket(io_service)`.

```
basic_datagram_socket(std::tr2::sys::io_service& io_service,
                      const protocol_type& protocol);
```

Effects: Constructs an object of class `basic_datagram_socket<Protocol, DatagramSocketService>`, initialising the base class with `basic_socket(io_service, protocol)`.

```
basic_datagram_socket(std::tr2::sys::io_service& io_service,
                      const endpoint_type& endpoint);
```

Effects: Constructs an object of class `basic_datagram_socket<Protocol, DatagramSocketService>`, initialising the base class with `basic_socket(io_service, endpoint)`.

```
basic_datagram_socket(std::tr2::sys::io_service& io_service,
                      const protocol_type& protocol,
                      const native_type& native_socket);
```

Effects: Constructs an object of class `basic_datagram_socket<Protocol, SocketService>`, initialising the base class with `basic_socket(io_service, protocol, native_socket)`.

5.7.8.2. `basic_datagram_socket` members

```
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers);
```

```
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 error_code& ec);
```

Returns: `this->service.receive(this->implementation, buffers, 0, 0, mutable_buffer(), ec)`.

```
template<class MutableBufferSequence, class ReadHandler>
  void async_receive(const MutableBufferSequence& buffers,
                     ReadHandler handler);
```

Effects: Calls `this->service.async_receive(this->implementation, buffers, 0, handler)`.

```
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags, error_code& ec);
```

Returns: `this->service.receive(this->implementation, buffers, flags, ec)`.

```
template<class MutableBufferSequence, class ReadHandler>
  void async_receive(const MutableBufferSequence& buffers,
                     socket_base::message_flags flags,
                     ReadHandler handler);
```

Effects: Calls `this->service.async_receive(this->implementation, buffers, flags, handler)`.

```
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender);
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender, error_code& ec);
```

Returns: `this->service.receive_from(this->implementation, buffers, sender, 0, ec)`.

```
template<class MutableBufferSequence, class ReadHandler>
  void async_receive_from(const MutableBufferSequence& buffers,
                          endpoint_type& sender,
                          ReadHandler handler);
```

Effects: Calls `this->service.async_receive_from(this->implementation, buffers, sender, 0, handler)`.

```
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender,
                      socket_base::message_flags flags);
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender,
                      socket_base::message_flags flags,
                      error_code& ec);
```

Returns: `this->service.receive_from(this->implementation, buffers, sender, flags, ec)`.

```
template<class MutableBufferSequence, class ReadHandler>
  void async_receive_from(const MutableBufferSequence& buffers,
                          endpoint_type& sender,
                          socket_base::message_flags flags,
                          ReadHandler handler);
```

Effects: Calls `this->service.async_receive_from(this->implementation, buffers, sender, flags, handler)`.

```
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers, error_code& ec);
```

Returns: `this->service.send(this->implementation, buffers, 0, ec)`.

```
template<class ConstBufferSequence, class WriteHandler>
  void async_send(const ConstBufferSequence& buffers, WriteHandler handler);
```

Effects: Calls `this->service.async_send(this->implementation, buffers, 0, handler)`.

```
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags, error_code& ec);
```

Returns: `this->service.send(this->implementation, buffers, flags, ec)`.

```
template<class ConstBufferSequence, class WriteHandler>
  void async_send(const ConstBufferSequence& buffers,
                  socket_base::message_flags flags,
                  WriteHandler handler);
```

Effects: Calls `this->service.async_send(this->implementation, buffers, flags, handler)`.

```
template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& destination);
template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& destination, error_code& ec);
```

Returns: `this->service.send_to(this->implementation, buffers, destination, 0, ec)`.

```
template<class ConstBufferSequence, class WriteHandler>
  void async_send_to(const ConstBufferSequence& buffers,
                     const endpoint_type& destination,
                     WriteHandler handler);
```

Effects: Calls `this->service.async_send_to(this->implementation, buffers, destination, 0, handler)`.

```
template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& destination,
                 socket_base::message_flags flags);
template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& destination,
                 socket_base::message_flags flags, error_code& ec);
```

Returns: `this->service.send_to(this->implementation, buffers, destination, flags, ec)`.

```
template<class ConstBufferSequence, class WriteHandler>
  void async_send_to(const ConstBufferSequence& buffers,
                     const endpoint_type& destination,
                     socket_base::message_flags flags,
                     WriteHandler handler);
```

Effects: Calls `this->service.async_send_to(this->implementation, buffers, destination, flags, handler)`.

5.7.9. Class template `stream_socket_service`

Instances of the `stream_socket_service` class template meet the requirements of a [`StreamSocketService`]().

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Protocol>
      class stream_socket_service :
        public io_service::service
      {
      public:
        static io_service::id id;

        // types:
        typedef Protocol protocol_type;
        typedef typename Protocol::endpoint endpoint_type;
        typedef unspecified implementation_type;
        typedef implementation defined native_type;
```

```
// constructors:
explicit stream_socket_service(io_service& ios);

// members:
void construct(implementation_type& impl);

void destroy(implementation_type& impl);

error_code open(implementation_type& impl,
                const protocol_type& protocol, error_code& ec);

error_code assign(implementation_type& impl,
                  const protocol_type& protocol,
                  const native_type& native_socket, error_code& ec);

bool is_open(const implementation_type& impl) const;

error_code close(implementation_type& impl, error_code& ec);

native_type native(implementation_type& impl);

error_code cancel(implementation_type& impl, error_code& ec);

template<class SettableSocketOption>
  error_code set_option(implementation_type& impl,
                        const SettableSocketOption& option,
                        error_code& ec);

template<class GettableSocketOption>
  error_code get_option(const implementation_type& impl,
                        GettableSocketOption& option,
                        error_code& ec) const;

template<class IoControlCommand>
  error_code io_control(implementation_type& impl,
                        IoControlCommand& command, error_code& ec);

bool at_mark(const implementation_type& impl, error_code& ec) const;

size_t available(const implementation_type& impl,
                 error_code& ec) const;

error_code bind(implementation_type& impl,
                const endpoint_type& endpoint, error_code& ec);

error_code shutdown(implementation_type& impl, shutdown_type how,
                    error_code& ec);

endpoint_type local_endpoint(const implementation_type& impl,
                             error_code& ec) const;

endpoint_type remote_endpoint(const implementation_type& impl,
                              error_code& ec) const;

error_code connect(implementation_type& impl,
                   const endpoint_type& endpoint, error_code& ec);

template<class ConnectHandler>
  void async_connect(implementation_type& impl,
                     const endpoint_type& endpoint,
                     ConnectHandler handler);

template<class MutableBufferSequence>
  size_t receive(implementation_type& impl,
                 const MutableBufferSequence& buffers,
                 socket_base::message_flags flags,
                 error_code& ec);

template<class MutableBufferSequence, class ReadHandler>
  void async_receive(implementation_type& impl,
                     const MutableBufferSequence& buffers,
                     socket_base::message_flags flags,
                     ReadHandler handler);

template<class ConstBufferSequence>
  size_t send(implementation_type& impl,
              const ConstBufferSequence& buffers,
              socket_base::message_flags flags,
```

```
                     error_code& ec);

        template<class ConstBufferSequence, class WriteHandler>
          void async_send(implementation_type& impl,
                          const ConstBufferSequence& buffers,
                          socket_base::message_flags flags,
                          WriteHandler handler);

      private:
        virtual void shutdown_service();
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.7.9.1. `stream_socket_service` types

`typedef` *`implementation defined`* `native_type;`

The native representation of a socket. Must satisfy the requirements of `CopyConstructible` types (C++ Std, 20.1.3), and the requirements of `Assignable` types (C++ Std, 23.1). [*Note:* For *POSIX* implementations, this type would typically be convertible to and from `int`. For *Windows* implementations, this type would typically be convertible to and from `SOCKET`. —*end note*]

5.7.9.2. `stream_socket_service` constructors

```
explicit stream_socket_service(io_service& ios);
```

Effects: Constructs an object of class `stream_socket_service<Protocol>`, initialising the base class with `io_service::service(ios)`.

5.7.9.3. `stream_socket_service` members

```
void shutdown_service();
```

Effects: Destroys all copies of user-defined handler objects owned by the service.

```
void construct(implementation_type& impl);
```

Effects: Initialises the socket implementation `impl`.

Postconditions: `!is_open(impl)`.

```
void destroy(implementation_type& impl);
```

Effects: If `is_open(impl)` is true, cancels pending asynchronous operations associated with `impl`, disables the linger socket option to prevent the operation from blocking, and releases socket resources as if by *POSIX* `close()`. Otherwise, no effect. Handlers for cancelled asynchronous operations are passed the `error_code` value `error::operation_aborted`. Failures are ignored.

```
error_code open(implementation_type& impl,
                const protocol_type& protocol, error_code& ec);
```

Requires: `!is_open(impl)`.

Effects: If `is_open(impl)` is true, sets `ec` to `error::already_open`. Otherwise establishes the postcondition, as if by *POSIX* `socket()`.

Returns: `ec`.

Postconditions: `is_open(impl)`.

```
error_code assign(implementation_type& impl,
                  const protocol_type& protocol,
                  const native_type& native_socket, error_code& ec);
```

Requires: `!is_open(impl)`.

Effects: If `is_open(impl)` is true, sets `ec` to `error::already_open`. Otherwise assigns the existing socket to the implementation `impl`.

Returns: `ec`.

Postconditions: `is_open(impl)`.

The main source of errors for `assign` would be a call to register the socket with an OS-specific event demultiplexor, such as a `kqueue`, an `epoll` descriptor, a `/dev/poll` device, or a Windows I/O completion port. These errors may also be produced by `open`, since that function would perform the same registration.

```
bool is_open(const implementation_type& impl) const;
```

Returns: A `bool` indicating whether the socket implementation `impl` was opened by a previous call to `open` or `assign`.

```
error_code close(implementation_type& impl, error_code& ec);
```

Effects: If `is_open(impl)` is true, cancels pending asynchronous operations associated with `impl`, and establishes the postcondition as if by *POSIX* [`close()`]. Otherwise, no effect. Handlers for cancelled asynchronous operations are passed the `error_code` value `error::operation_aborted`.

Returns: `ec`.

```
native_type native(implementation_type& impl);
```

Returns: The native representation of the socket implementation `impl`.

```
error_code cancel(implementation_type& impl, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise cancels pending asynchronous operations associated with `impl`, if any. Handlers for cancelled asynchronous operations are passed the `error_code` value `error::operation_aborted`.

The conditions under which cancellation of asynchronous operations is permitted are implementation-defined. If current conditions do not permit cancellation, an implementation shall set `ec` to `error::operation_not_supported`.

This flexibility is included to support implementations on Windows versions prior to Vista, where the `CancelIo` function will only cancel asynchronous operations started from the same thread. Vista provides `CancelIoEx` which may be used to cancel all asynchronous operations associated with a socket.

Returns: `ec`.

```
template<class SettableSocketOption>
  error_code set_option(implementation_type& impl,
                        const SettableSocketOption& option,
                        error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise sets an option on the socket `impl`, as if by *POSIX* [`setsockopt()`].

Returns: `ec`.

```
template<class GettableSocketOption>
  error_code get_option(const implementation_type& impl,
                        GettableSocketOption& option,
                        error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise gets an option from the socket `impl`, as if by *POSIX* [`getsockopt()`].

Returns: `ec`.

```
template<class IoControlCommand>
  error_code io_control(implementation_type& impl,
                        IoControlCommand& command, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise executes an I/O control command on the socket `impl`, as if by *POSIX* `ioctl()`.

Returns: `ec`.

This proposal does not include any classes that satisfy `IoControlCommand` requirements. However, implementation-specific extensions such as QoS may be implemented using `ioctl()`, and the `io_control()` operation is included to allow these extensions to be supported.

```
bool at_mark(const implementation_type& impl, error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. If the value returned by `native(impl)` is not a valid socket, sets `ec` to `error::not_socket`. [*Note:* Instead of *POSIX* `ENOTTY`. —*end note*] Otherwise determines if the socket `impl` is at the out-of-band data mark, as if by *POSIX* `sockatmark()`.

Returns: A `bool` indicating whether the socket is at the out-of-band data mark.

```
size_t available(const implementation_type& impl,
                 error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise determines the number of bytes that may be read without blocking.

Returns: The number of bytes that may be read without blocking, or 0 if an error occurs.

```
error_code bind(implementation_type& impl,
                const endpoint_type& endpoint, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise binds the socket `impl` to the specified local endpoint, as if by *POSIX* `bind()`.

Returns: `ec`.

```
error_code shutdown(implementation_type& impl, shutdown_type how,
                    error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise shuts down all or part of a full-duplex connection for the socket `impl`, as if by *POSIX* `shutdown()`. If `how` is `shutdown_receive`, uses *POSIX* `SHUT_RD`. If `how` is `shutdown_send`, uses *POSIX* `SHUT_WR`. If `how` is `shutdown_both`, uses *POSIX* `SHUT_RDWR`.

Returns: `ec`.

```
endpoint_type local_endpoint(const implementation_type& impl,
                             error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise determines the locally-bound endpoint associated with the socket `impl`, as if by *POSIX* `getsockname()`.

Returns: On success, the locally-bound endpoint associated with `impl`. Otherwise `endpoint_type()`.

```
endpoint_type remote_endpoint(const implementation_type& impl,
                              error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise determines the remote endpoint associated with the socket `impl`, as if by *POSIX* `getpeername()`.

Returns: On success, the remote endpoint associated with `impl`. Otherwise `endpoint_type()`.

```
error_code connect(implementation_type& impl,
```

```
                    const endpoint_type& endpoint, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise connects the socket `impl` to the specified remote endpoint, as if by *POSIX* `connect()`.

Returns: `ec`.

```
template<class ConnectHandler>
  void async_connect(implementation_type& impl,
                     const endpoint_type& endpoint,
                     ConnectHandler handler);
```

Requires: `is_open(impl)`.

Effects: Initiates an asynchronous operation to connect the socket `impl` to the specified remote endpoint, as if by *POSIX* `connect()`. The operation is performed via the `io_service` object returned by `io_service::service::io_service()` and behaves according to asynchronous operation requirements.

If `is_open(impl)` is false, the operation shall fail immediately with `error_code` value `error::bad_descriptor`.

When multiple asynchronous connect operations are initiated such that the operations may logically be performed in parallel, the behaviour is implementation-defined. When an asynchronous connect operation and an asynchronous read or write operation are initiated such that they may logically be performed in parallel, the behaviour is implementation-defined.

If a program performs a synchronous `set_option`, `io_control`, `bind`, `shutdown`, `connect`, `receive`, or `send` operation on `impl` while there is an incomplete asynchronous connect operation, the behaviour is implementation-defined.

```
template<class MutableBufferSequence>
  size_t receive(implementation_type& impl,
                 const MutableBufferSequence& buffers,
                 socket_base::message_flags flags,
                 error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise reads data from a connected socket `impl`, as if by *POSIX* `recvmsg()`.

The mutable buffer sequence `buffers` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next. If the total size of all buffers in the sequence `buffers` is `0`, the function shall return `0` immediately.

The `flags` argument specifies the type of receive operation to be performed.

If the operation completes due to graceful connection closure by the peer, the operation shall fail with `error::eof`.

Returns: On success, the number of bytes read. Otherwise `0`.

```
template<class MutableBufferSequence, class ReadHandler>
  void async_receive(implementation_type& impl,
                     const MutableBufferSequence& buffers,
                     socket_base::message_flags flags,
                     ReadHandler handler);
```

Requires: `is_open(impl) && (flags & socket_base::message_peek) == 0`.

Effects: Initiates an asynchronous operation to read data from a connected socket `impl`, as if by *POSIX* `recvmsg()`. The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements.

If `is_open(impl)` is false, the operation shall fail immediately with `error_code` value `error::bad_descriptor`.

The mutable buffer sequence `buffers` specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `buffers` until such time as the read operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until:

— the last copy of `buffers` is destroyed, or

— the handler for the asynchronous read operation is invoked,

whichever comes first. If the total size of all buffers in the sequence `buffers` is `0`, the asynchronous read operation shall complete immediately and pass `0` as the argument to the handler that specifies the number of bytes read.

The `flags` argument specifies the type of receive operation to be performed. If `flags & socket_base::message_peek` is true, the operation shall fail immediately with `error::invalid_argument`. If `flags & socket_base::message_out_of_band` is true, the operation shall continue until out-of-band data is received, or an error occurs.

When multiple asynchronous read operations with zero `flags` are initiated such that the operations may logically be performed in parallel, the implementation shall fill the `buffers` in the order in which the operations were issued. The order of invocation of the handlers for these operations is implementation-defined. When multiple asynchronous read operations with non-zero `flags` are initiated such that operations may logically be performed in parallel, the behaviour is implementation-defined.

If a program performs a synchronous `set_option`, `io_control`, `bind`, `shutdown`, `connect`, or `receive` operation on `impl` while there is an incomplete asynchronous read operation, the behaviour is implementation-defined.

If the operation completes due to graceful connection closure by the peer, the operation shall fail with `error::eof`.

If the operation completes successfully, the `ReadHandler` object `handler` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.

```
template<class ConstBufferSequence>
  size_t send(implementation_type& impl,
              const ConstBufferSequence& buffers,
              socket_base::message_flags flags,
              error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise writes data to a connected socket `impl`, as if by *POSIX* sendmsg().

The constant buffer sequence `buffers` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next. If the total size of all buffers in the sequence `buffers` is `0`, the function shall return `0` immediately.

The `flags` argument specifies the type of send operation to be performed.

Returns: On success, the number of bytes written. Otherwise `0`.

```
template<class ConstBufferSequence, class WriteHandler>
  void async_send(implementation_type& impl,
                  const ConstBufferSequence& buffers,
                  socket_base::message_flags flags,
                  WriteHandler handler);
```

Requires: `is_open(impl)`.

Effects: Initiates an asynchronous operation to write data to a connected socket `impl`, as if by *POSIX* sendmsg(). The operation is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements.

If `is_open(impl)` is false, the operation shall fail immediately with `error_code` value `error::bad_descriptor`.

The constant buffer sequence `buffers` specifies memory where the data to be written is located. The operation shall always write a buffer in the sequence completely before proceeding to the next. The implementation shall maintain one or more copies of `buffers` until such time as the write operation no longer requires access to the memory specified by the buffers in the sequence. The program must ensure the memory is valid until:

— the last copy of `buffers` is destroyed, or

— the handler for the asynchronous write operation is invoked,

whichever comes first. If the total size of all buffers in the sequence `buffers` is `0`, the asynchronous write operation shall complete immediately and pass `0` as the argument to the handler that specifies the number of bytes written.

The `flags` argument specifies the type of send operation to be performed.

When multiple asynchronous write operations with zero `flags` are initiated such that the operations may logically be performed in parallel, the implementation shall write the `buffers` in the order in which the operations were issued. The order of invocation of the handlers for these operations is implementation-defined. When multiple asynchronous write operations with non-zero `flags` are initiated such that operations may logically be performed in parallel, the behaviour is implementation-defined.

If a program performs a synchronous `set_option`, `io_control`, `bind`, `shutdown`, `connect`, or `send` operation on `impl` while there is an incomplete asynchronous send operation, the behaviour is implementation-defined.

If the operation completes successfully, the `WriteHandler` object `handler` is invoked with the number of bytes transferred. Otherwise it is invoked with `0`.

5.7.10. Class template `basic_stream_socket`

Instances of the `basic_stream_socket` class template meet the requirements for synchronous read streams, synchronous write streams, asynchronous read streams, and asynchronous write streams.

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Protocol, class StreamSocketService>
      class basic_stream_socket :
        public basic_socket<StreamSocketService>
      {
      public:
        // types:
        typedef typename StreamSocketService::native_type native_type;
        typedef Protocol protocol_type;
        typedef typename Protocol::endpoint endpoint_type;

        // constructors:
        explicit basic_stream_socket(std::tr2::sys::io_service& io_service);
        basic_stream_socket(std::tr2::sys::io_service& io_service,
                            const protocol_type& protocol);
        basic_stream_socket(std::tr2::sys::io_service& io_service,
                            const endpoint_type& endpoint);
        basic_stream_socket(std::tr2::sys::io_service& io_service,
                            const protocol_type& protocol,
                            const native_type& native_socket);

        // members:
        template<class MutableBufferSequence>
          size_t receive(const MutableBufferSequence& buffers);
        template<class MutableBufferSequence>
          size_t receive(const MutableBufferSequence& buffers,
                         error_code& ec);

        template<class MutableBufferSequence, class ReadHandler>
          void async_receive(const MutableBufferSequence& buffers,
                             ReadHandler handler);

        template<class MutableBufferSequence>
          size_t receive(const MutableBufferSequence& buffers,
                         socket_base::message_flags flags);
        template<class MutableBufferSequence>
          size_t receive(const MutableBufferSequence& buffers,
                         socket_base::message_flags flags, error_code& ec);

        template<class MutableBufferSequence, class ReadHandler>
          void async_receive(const MutableBufferSequence& buffers,
```

```
                             socket_base::message_flags flags,
                             ReadHandler handler);

    template<class ConstBufferSequence>
      size_t send(const ConstBufferSequence& buffers);
    template<class ConstBufferSequence>
      size_t send(const ConstBufferSequence& buffers, error_code& ec);

    template<class ConstBufferSequence, class WriteHandler>
      void async_send(const ConstBufferSequence& buffers,
                      WriteHandler handler);

    template<class ConstBufferSequence>
      size_t send(const ConstBufferSequence& buffers,
                  socket_base::message_flags flags);
    template<class ConstBufferSequence>
      size_t send(const ConstBufferSequence& buffers,
                  socket_base::message_flags flags, error_code& ec);

    template<class ConstBufferSequence, class WriteHandler>
      void async_send(const ConstBufferSequence& buffers,
                      socket_base::message_flags flags,
                      WriteHandler handler);

    template<class MutableBufferSequence>
      size_t read_some(const MutableBufferSequence& buffers);
    template<class MutableBufferSequence>
      size_t read_some(const MutableBufferSequence& buffers,
                       error_code& ec);

    template<class MutableBufferSequence, class ReadHandler>
      void async_read_some(const MutableBufferSequence& buffers,
                           ReadHandler handler);

    template<class ConstBufferSequence>
      size_t write_some(const ConstBufferSequence& buffers);
    template<class ConstBufferSequence>
      size_t write_some(const ConstBufferSequence& buffers,
                        error_code& ec);

    template<class ConstBufferSequence, class WriteHandler>
      void async_write_some(const ConstBufferSequence& buffers,
                            WriteHandler handler);
  };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.7.10.1. `basic_stream_socket` constructors

```
explicit basic_stream_socket(std::tr2::sys::io_service& io_service);
```

Effects: Constructs an object of class `basic_stream_socket<Protocol, StreamSocketService>`, initialising the base class with `basic_socket(io_service)`.

```
basic_stream_socket(std::tr2::sys::io_service& io_service,
                    const protocol_type& protocol);
```

Effects: Constructs an object of class `basic_stream_socket<Protocol, StreamSocketService>`, initialising the base class with `basic_socket(io_service, protocol)`.

```
basic_stream_socket(std::tr2::sys::io_service& io_service,
                    const endpoint_type& endpoint);
```

Effects: Constructs an object of class `basic_stream_socket<Protocol, StreamSocketService>`, initialising the base class with `basic_socket(io_service, endpoint)`.

```
basic_stream_socket(std::tr2::sys::io_service& io_service,
                    const protocol_type& protocol,
                    const native_type& native_socket);
```

Effects: Constructs an object of class `basic_stream_socket<Protocol, SocketService>`, initialising the base class with `basic_socket(io_service, protocol, native_socket)`.

5.7.10.2. `basic_stream_socket` members

```
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 error_code& ec);
```

Returns: `this->service.receive(this->implementation, buffers, 0, 0, mutable_buffer(), ec)`.

```
template<class MutableBufferSequence, class ReadHandler>
  void async_receive(const MutableBufferSequence& buffers,
                     ReadHandler handler);
```

Effects: Calls `this->service.async_receive(this->implementation, buffers, 0, handler)`.

```
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags, error_code& ec);
```

Returns: `this->service.receive(this->implementation, buffers, flags, ec)`.

```
template<class MutableBufferSequence, class ReadHandler>
  void async_receive(const MutableBufferSequence& buffers,
                     socket_base::message_flags flags,
                     ReadHandler handler);
```

Effects: Calls `this->service.async_receive(this->implementation, buffers, flags, handler)`.

```
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers, error_code& ec);
```

Returns: `this->service.send(this->implementation, buffers, 0, ec)`.

```
template<class ConstBufferSequence, class WriteHandler>
  void async_send(const ConstBufferSequence& buffers, WriteHandler handler);
```

Effects: Calls `this->service.async_send(this->implementation, buffers, 0, handler)`.

```
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags, error_code& ec);
```

Returns: `this->service.send(this->implementation, buffers, flags, ec)`.

```
template<class ConstBufferSequence, class WriteHandler>
  void async_send(const ConstBufferSequence& buffers,
                  socket_base::message_flags flags,
                  WriteHandler handler);
```

Effects: Calls `this->service.async_send(this->implementation, buffers, flags, handler)`.

```
template<class MutableBufferSequence>
  size_t read_some(const MutableBufferSequence& buffers);
template<class MutableBufferSequence>
  size_t read_some(const MutableBufferSequence& buffers,
                   error_code& ec);
```

Returns: `this->service.receive(this->implementation, buffers, 0, 0, mutable_buffer(), ec)`.

```
template<class MutableBufferSequence, class ReadHandler>
  void async_read_some(const MutableBufferSequence& buffers,
                       ReadHandler handler);
```

Effects: Calls `this->service.async_receive(this->implementation, buffers, 0, handler)`.

```
template<class ConstBufferSequence>
  size_t write_some(const ConstBufferSequence& buffers);
template<class ConstBufferSequence>
  size_t write_some(const ConstBufferSequence& buffers,
                    error_code& ec);
```

Returns: `this->service.send(this->implementation, buffers, 0, ec)`.

```
template<class ConstBufferSequence, class WriteHandler>
  void async_write_some(const ConstBufferSequence& buffers,
                        WriteHandler handler);
```

Effects: Calls `this->service.async_send(this->implementation, buffers, 0, handler)`.

5.7.11. Class template `socket_acceptor_service`

Instances of the `socket_acceptor_service` class template meet the requirements of a [SocketAcceptorService](#).

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Protocol>
      class socket_acceptor_service :
        public io_service::service
      {
      public:
        static io_service::id id;

        // types:
        typedef Protocol protocol_type;
        typedef typename Protocol::endpoint endpoint_type;
        typedef unspecified implementation_type;
        typedef implementation defined native_type;

        // constructors:
        explicit socket_acceptor_service(io_service& ios);

        // members:
        void construct(implementation_type& impl);

        void destroy(implementation_type& impl);

        error_code open(implementation_type& impl,
                        const protocol_type& protocol, error_code& ec);

        error_code assign(implementation_type& impl,
                          const protocol_type& protocol,
                          const native_type& native_socket, error_code& ec);

        bool is_open(const implementation_type& impl) const;

        error_code close(implementation_type& impl, error_code& ec);

        native_type native(implementation_type& impl);

        error_code cancel(implementation_type& impl, error_code& ec);

        template<class SettableSocketOption>
          error_code set_option(implementation_type& impl,
                                const SettableSocketOption& option,
                                error_code& ec);

        template<class GettableSocketOption>
          error_code get_option(const implementation_type& impl,
                                GettableSocketOption& option,
                                error_code& ec) const;

        template<class IoControlCommand>
          error_code io_control(implementation_type& impl,
                                IoControlCommand& command, error_code& ec);

        error_code bind(implementation_type& impl,
                        const endpoint_type& endpoint, error_code& ec);

        error_code listen(implementation_type& impl, int backlog,
                          error_code& ec);
```

```
        endpoint_type local_endpoint(const implementation_type& impl,
                                     error_code& ec) const;

        template<class SocketService>
          error_code accept(implementation_type& impl,
                            basic_socket<Protocol, SocketService>& socket,
                            endpoint_type* endpoint, error_code& ec);

        template<class SocketService, class AcceptHandler>
          void async_accept(implementation_type& impl,
                            basic_socket<Protocol, SocketService>& socket,
                            endpoint_type* endpoint, AcceptHandler handler);

      private:
        virtual void shutdown_service();
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.7.11.1. `socket_acceptor_service` types

`typedef` *`implementation defined`* `native_type;`

> The native representation of a socket acceptor. Must satisfy the requirements of `CopyConstructible` types (C++ Std, 20.1.3), and the requirements of `Assignable` types (C++ Std, 23.1). [*Note:* For *POSIX* implementations, this type would typically be convertible to and from `int`. For *Windows* implementations, this type would typically be convertible to and from `SOCKET`. —*end note*]

5.7.11.2. `socket_acceptor_service` constructors

`explicit socket_acceptor_service(io_service& ios);`

> *Effects:* Constructs an object of class `socket_acceptor_service<Protocol>`, initialising the base class with `io_service::service(ios)`.

5.7.11.3. `socket_acceptor_service` members

`void shutdown_service();`

> *Effects:* Destroys all copies of user-defined handler objects owned by the service.

`void construct(implementation_type& impl);`

> *Effects:* Initialises the socket acceptor implementation `impl`.
>
> *Postconditions:* `!is_open(impl)`.

`void destroy(implementation_type& impl);`

> *Effects:* If `is_open(impl)` is true, cancels pending asynchronous operations associated with `impl`, and releases socket acceptor resources as if by *POSIX* `close()`. Otherwise, no effect. Handlers for cancelled asynchronous operations are passed the `error_code` value `error::operation_aborted`. Failures are ignored.

```
error_code open(implementation_type& impl,
                const protocol_type& protocol, error_code& ec);
```

> *Requires:* `!is_open(impl)`.
>
> *Effects:* If `is_open(impl)` is true, sets `ec` to `error::already_open`. Otherwise establishes the postcondition, as if by *POSIX* `socket()`.
>
> *Returns:* `ec`.
>
> *Postconditions:* `is_open(impl)`.

```
error_code assign(implementation_type& impl,
                  const protocol_type& protocol,
                  const native_type& native_socket, error_code& ec);
```

> *Requires:* `!is_open(impl)`.

Effects: If `is_open(impl)` is true, sets `ec` to `error::already_open`. Otherwise assigns the existing socket acceptor to the implementation `impl`.

Returns: `ec`.

Postconditions: `is_open(impl)`.

The main source of errors for `assign` would be a call to register the socket acceptor with an OS-specific event demultiplexor, such as a `kqueue`, an `epoll` descriptor, a `/dev/poll` device, or a Windows I/O completion port. These errors may also be produced by `open`, since that function would perform the same registration.

```
bool is_open(const implementation_type& impl) const;
```

Returns: A `bool` indicating whether the socket acceptor implementation `impl` was opened by a previous call to `open` or `assign`.

```
error_code close(implementation_type& impl, error_code& ec);
```

Effects: If `is_open(impl)` is true, cancels pending asynchronous operations associated with `impl`, and establishes the postcondition as if by *POSIX* `close()`. Otherwise, no effect. Handlers for cancelled asynchronous operations are passed the `error_code` value `error::operation_aborted`.

Returns: `ec`.

Postconditions: `!is_open(impl)`.

```
native_type native(implementation_type& impl);
```

Returns: The native representation of the socket acceptor implementation `impl`.

```
error_code cancel(implementation_type& impl, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise cancels pending asynchronous operations associated with `impl`, if any. Handlers for cancelled asynchronous operations are passed the `error_code` value `error::operation_aborted`.

The conditions under which cancellation of asynchronous operations is permitted are implementation-defined. If current conditions do not permit cancellation, an implementation shall set `ec` to `error::operation_not_supported`.

This flexibility is included to support implementations on Windows versions prior to Vista, where the `CancelIo` function will only cancel asynchronous operations started from the same thread. Vista provides `CancelIoEx` which may be used to cancel all asynchronous operations associated with a socket.

Returns: `ec`.

```
template<class SettableSocketOption>
  error_code set_option(implementation_type& impl,
                        const SettableSocketOption& option,
                        error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise sets an option on the socket acceptor `impl`, as if by *POSIX* `setsockopt()`.

Returns: `ec`.

```
template<class GettableSocketOption>
  error_code get_option(const implementation_type& impl,
                        GettableSocketOption& option,
                        error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise gets an option from the socket acceptor `impl`, as if by *POSIX* `getsockopt()`.

Returns: `ec`.

```
template<class IoControlCommand>
  error_code io_control(implementation_type& impl,
                        IoControlCommand& command, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise executes an I/O control command on the socket acceptor `impl`, as if by *POSIX* `ioctl()`.

Returns: `ec`.

This proposal does not include any classes that satisfy `IoControlCommand` requirements. However, implementation-specific extensions such as QoS may be implemented using `ioctl()`, and the `io_control()` operation is included to allow these extensions to be supported.

```
error_code bind(implementation_type& impl,
                const endpoint_type& endpoint, error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise binds the socket `impl` to the specified local endpoint, as if by *POSIX* `bind()`.

Returns: `ec`.

```
error_code listen(implementation_type& impl, int backlog,
                  error_code& ec);
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwse marks the socket acceptor `impl` as ready to accept connections, as if by *POSIX* `listen()`. If `backlog == socket_base::max_connections`, the implementation shall set the socket acceptor's listen queue to the maximum allowable length.

Returns: `ec`.

```
endpoint_type local_endpoint(const implementation_type& impl,
                             error_code& ec) const;
```

Requires: `is_open(impl)`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. Otherwise determines the locally-bound endpoint associated with the socket acceptor `impl`, as if by *POSIX* `getsockname()`.

Returns: On success, the locally-bound endpoint associated with `impl`. Otherwise `endpoint_type()`.

```
template<class SocketService>
  error_code accept(implementation_type& impl,
                    basic_socket<Protocol, SocketService>& socket,
                    endpoint_type* endpoint, error_code& ec);
```

Requires: `is_open(impl) && !socket.is_open()`, and a prior call to `listen` for the socket acceptor implementation `impl`.

Effects: If `is_open(impl)` is false, sets `ec` to `error::bad_descriptor`. If `socket.is_open()` is true, sets `ec` to `error::already_open`. Otherwise associates `socket` with the first connection extracted from the queue of pending connections of the socket acceptor `impl`, as if by *POSIX* `accept()`. On success, and if `endpoint != 0`, and if `endpoint != 0`, assigns into `*endpoint` the remote endpoint of the connection.

Implementations shall not permit the operation to complete with an error indicating that a connection has been aborted. [*Note:* Such as *POSIX* `ECONNABORTED`, and *Windows* `WSAECONNABORTED` or `ERROR_NETNAME_DELETED`). *—end note*]

The `ECONNABORTED` error is suppressed since it is a transient error, is rarely (if ever) useful but handling it correctly adds complexity to even very simple servers.

Returns: `ec`.

Postconditions: `socket.is_open()`.

```
template<class SocketService, class AcceptHandler>
  void async_accept(implementation_type& impl,
                    basic_socket<Protocol, SocketService>& socket,
                    endpoint_type* endpoint, AcceptHandler handler);
```

Requires: `is_open(impl) && !socket.is_open()`, and a prior call to `listen` for the socket acceptor implementation `impl`.

Effects: Initiates an asynchronous operation to associate `socket` with the first connection extracted from the queue of pending connections of the socket acceptor `impl`, as if by *POSIX* `accept()`. The operation is performed via the `io_service` object returned by `io_service::service::io_service()` and behaves according to asynchronous operation requirements.

If `is_open(impl)` is false, the operation shall fail immediately with `error_code` value `error::bad_descriptor`. If `socket.is_open()` is true, the operation shall fail immediately with `error_code` value `error::already_open`.

When multiple asynchronous accept operations are initiated such that the operations may logically be performed in parallel, the behaviour is implementation-defined.

If a program performs a synchronous `set_option`, `io_control`, `bind`, or `accept` operation on `impl` while there is an incomplete asynchronous accept operation, the behaviour is implementation-defined.

The program must ensure the `basic_socket<>` object `socket` is valid until the handler for the asynchronous operation is invoked. If `endpoint != 0`, the program must ensure the object `*endpoint` is valid until the handler for the asynchronous operation is invoked.

Implementations shall not permit the operation to complete with an error indicating that a connection has been aborted. [*Note:* Such as *POSIX* `ECONNABORTED`, and *Windows* `WSAECONNABORTED` or `ERROR_NETNAME_DELETED`). *—end note*]

5.7.12. Class template `basic_socket_acceptor`

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Protocol, class SocketAcceptorService>
      class basic_socket_acceptor :
        public basic_io_object<SocketAcceptorService>,
        public socket_base
      {
      public:
        // types:
        typedef typename SocketAcceptorService::native_type native_type;
        typedef Protocol protocol_type;
        typedef typename Protocol::endpoint endpoint_type;

        // constructors:
        explicit basic_socket_acceptor(std::tr2::sys::io_service& io_service);
        basic_socket_acceptor(std::tr2::sys::io_service& io_service,
                              const protocol_type& protocol);
        basic_socket_acceptor(std::tr2::sys::io_service& io_service,
                              const endpoint_type& endpoint,
                              bool reuse_addr = true);
        basic_socket_acceptor(std::tr2::sys::io_service& io_service,
                              const protocol_type& protocol,
                              const native_type& native_acceptor);

        // members:
        native_type native();

        void open(const protocol_type& protocol = protocol_type());
        error_code open(const protocol_type& protocol, error_code& ec);

        void assign(const protocol_type& protocol,
                    const native_type& native_acceptor);
        error_code assign(const protocol_type& protocol,
                          const native_type& native_acceptor, error_code& ec);

        bool is_open() const;

        void close();
```

```
    error_code close(error_code& ec);

    void cancel();
    error_code cancel(error_code& ec);

    template<class SettableSocketOption>
      void set_option(const SettableSocketOption& option);
    template<class SettableSocketOption>
      error_code set_option(const SettableSocketOption& option,
                            error_code& ec);

    template<class GettableSocketOption>
      void get_option(GettableSocketOption& option) const;
    template<class GettableSocketOption>
      error_code get_option(GettableSocketOption& option,
                            error_code& ec) const;

    template<class IoControlCommand>
      void io_control(IoControlCommand& command);
    template<class IoControlCommand>
      error_code io_control(IoControlCommand& command, error_code& ec);

    void bind(const endpoint_type& endpoint);
    error_code bind(const endpoint_type& endpoint, error_code& ec);

    void listen(int backlog = max_connections);
    error_code listen(int backlog, error_code& ec);

    endpoint_type local_endpoint() const;
    endpoint_type local_endpoint(error_code& ec) const;

    template<class SocketService>
      void accept(basic_socket<Protocol, SocketService>& socket);
    template<class SocketService>
      error_code accept(basic_socket<Protocol, SocketService>& socket,
                        error_code& ec);

    template<class SocketService, class AcceptHandler>
      void async_accept(basic_socket<Protocol, SocketService>& socket,
                        AcceptHandler handler);

    template<class SocketService>
      void accept(basic_socket<Protocol, SocketService>& socket,
                  endpoint_type& endpoint);
    template<class SocketService>
      error_code accept(basic_socket<Protocol, SocketService>& socket,
                        endpoint_type& endpoint, error_code& ec);

    template<class SocketService, class AcceptHandler>
      void async_accept(basic_socket<Protocol, SocketService>& socket,
                        endpoint_type& endpoint, AcceptHandler handler);
  };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.7.12.1. `basic_socket_acceptor` constructors

```
explicit basic_socket_acceptor(std::tr2::sys::io_service& io_service);
```

Effects: Constructs an object of class `basic_socket_acceptor<Protocol, SocketAcceptorService>`, initialising the base class with `basic_io_object(io_service)`.

```
basic_socket_acceptor(std::tr2::sys::io_service& io_service,
                      const protocol_type& protocol);
```

Effects: Constructs an object of class `basic_socket_acceptor<Protocol, SocketAcceptorService>`, initialising the base class with `basic_io_object(io_service)`, then opening the socket as if by calling:

```
error_code ec;
this->service.open(this->implementation, protocol, ec);
if (ec) throw system_error(ec);
```

```
basic_socket_acceptor(std::tr2::sys::io_service& io_service,
                      const endpoint_type& endpoint,
                      bool reuse_addr = true);
```

Effects: Constructs an object of class `basic_socket_acceptor<Protocol, SocketAcceptorService>`, initialising the base class with `basic_io_object(io_service)`, then opening and binding the socket and marking it as ready to accept connections as if by calling:

```
error_code ec;
this->service.open(this->implementation, endpoint.protocol(), ec);
if (ec) throw system_error(ec);
if (reuse_addr)
{
  this->service.set_option(this->implementation, reuse_address(true), ec);
  if (ec) throw system_error(ec);
}
this->service.bind(this->implementation, endpoint, ec);
if (ec) throw system_error(ec);
this->service.listen(this->implementation, max_connections, ec);
if (ec) throw system_error(ec);
```

[UNPV1] recommends setting `SO_REUSEADDR` by default in all TCP servers, although there are some security implications in doing so.

```
basic_socket_acceptor(std::tr2::sys::io_service& io_service,
                      const protocol_type& protocol,
                      const native_type& native_acceptor);
```

Effects: Constructs an object of class `basic_socket_acceptor<Protocol, SocketAcceptorService>`, initialising the base class with `basic_io_object(io_service)`, then assigning the existing native acceptor into the object as if by calling:

```
error_code ec;
this->service.assign(this->implementation, protocol, native_acceptor, ec);
if (ec) throw system_error(ec);
```

5.7.12.2. `basic_socket_acceptor` members

```
native_type native();
```

Returns: `this->service.native(this->implementation)`.

```
void open(const protocol_type& protocol);
error_code open(const protocol_type& protocol, error_code& ec);
```

Returns: `this->service.open(this->implementation, protocol, ec)`.

```
void assign(const protocol_type& protocol,
            const native_type& native_acceptor);
error_code assign(const protocol_type& protocol,
                  const native_type& native_acceptor, error_code& ec);
```

Returns: `this->service.assign(this->implementation, protocol, native_acceptor, ec)`.

```
bool is_open() const;
```

Returns: `this->service.is_open(this->implementation)`.

```
void close();
error_code close(error_code& ec);
```

Returns: `this->service.close(this->implementation, ec)`.

```
void cancel();
error_code cancel(error_code& ec);
```

Returns: `this->service.cancel(this->implementation, ec)`.

```
template<class SettableSocketOption>
  void set_option(const SettableSocketOption& option);
template<class SettableSocketOption>
  error_code set_option(const SettableSocketOption& option,
                        error_code& ec);
```

Returns: `this->service.set_option(this->implementation, option, ec)`.

```
template<class GettableSocketOption>
  void get_option(GettableSocketOption& option);
template<class GettableSocketOption>
  error_code get_option(GettableSocketOption& option, error_code& ec);
```

Returns: `this->service.get_option(this->implementation, option, ec)`.

```
template<class IoControlCommand>
  void io_control(IoControlCommand& command);
template<class IoControlCommand>
  error_code io_control(IoControlCommand& command, error_code& ec);
```

Returns: `this->service.io_control(this->implementation, command, ec)`.

```
void bind(const endpoint_type& endpoint);
error_code bind(const endpoint_type& endpoint, error_code& ec);
```

Returns: `this->service.bind(this->implementation, endpoint, ec)`.

```
void listen(int backlog = max_connections);
error_code listen(int backlog, error_code& ec);
```

Returns: `this->service.listen(this->implementation, backlog, ec)`.

```
endpoint_type local_endpoint() const;
endpoint_type local_endpoint(error_code& ec) const;
```

Returns: `this->service.local_endpoint(this->implementation, ec)`.

```
template<class SocketService>
  void accept(basic_socket<Protocol, SocketService>& socket);
template<class SocketService>
  error_code accept(basic_socket<Protocol, SocketService>& socket,
                    error_code& ec);
```

Returns: `this->service.accept(this->implementation, socket, 0, ec)`.

```
template<class SocketService, class AcceptHandler>
  void async_accept(basic_socket<Protocol, SocketService>& socket,
                    AcceptHandler handler);
```

Effects: Calls `this->service.async_accept(this->implementation, socket, 0, handler)`.

```
template<class SocketService>
  void accept(basic_socket<Protocol, SocketService>& socket,
              endpoint_type& endpoint);
template<class SocketService>
  error_code accept(basic_socket<Protocol, SocketService>& socket,
                    endpoint_type& endpoint, error_code& ec);
```

Returns: `this->service.accept(this->implementation, socket, &endpoint, ec)`.

```
template<class SocketService, class AcceptHandler>
  void async_accept(basic_socket<Protocol, SocketService>& socket,
                    endpoint_type& endpoint, AcceptHandler handler);
```

Effects: Calls `this->service.async_accept(this->implementation, socket, &endpoint, handler)`.

5.8. Socket streams

5.8.1. Class template `basic_socket_streambuf`

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Protocol, class StreamSocketService>
      class basic_socket_streambuf :
        public basic_streambuf<char>,
        public basic_socket<Protocol, StreamSocketService>
      {
      public:
        // types:
        typedef typename Protocol::endpoint endpoint_type;

        // constructors/destructor:
        basic_socket_streambuf();
        virtual ~basic_socket_streambuf();
```

```
        // members:
        basic_socket_streambuf<Protocol, StreamSocketService>*
          connect(const endpoint_type& e);
        template<class T1, class T2, ..., class TN>
          basic_socket_streambuf<Protocol, StreamSocketService>*
            connect(T1 t1, T2 t2, ..., TN tN);

        basic_socket_streambuf<Protocol, StreamSocketService>* close();

      protected:
        // overridden virtual functions:
        virtual int_type underflow();
        virtual int_type pbackfail(int_type c = traits_type::eof());
        virtual int_type overflow(int_type c = traits_type::eof());
        virtual int sync();
        virtual streambuf* setbuf(char_type* s, streamsize n);
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

The class `basic_socket_streambuf<Protocol, StreamSocketService>` associates both the input sequence and the output sequence with a socket. The input and output sequences are independent and do not support seeking. Multibyte/wide character conversion is not supported.

5.8.1.1. `basic_socket_streambuf` constructors

```
basic_socket_streambuf();
```

Effects: Constructs an object of class `basic_socket_streambuf<Protocol, StreamSocketService>`, initialising the base classes with `basic_streambuf<char>()` and `basic_socket<Protocol, StreamSocketService>(ios)`, where `ios` is an implementation-defined object of class `io_service` that has a longer lifetime than the `basic_socket<Protocol, StreamSocketService>` base.

```
virtual ~basic_socket_streambuf();
```

Effects: Destroys an object of class `basic_socket_streambuf<Protocol, StreamSocketService>`. If a put area exists, calls `overflow(traits_type::eof())` to flush characters. [*Note:* The socket is closed by the `basic_socket<Protocol, StreamSocketService>` destructor. *—end note*]

5.8.1.2. `basic_socket_streambuf` members

```
basic_socket_streambuf<Protocol, StreamSocketService>*
  connect(const endpoint_type& e);
```

Effects: Resets the `streambuf` get and put areas, then opens a connection as if by calling:

```
error_code ec;
this->basic_socket<Protocol, StreamSocketService>::close(ec);
this->basic_socket<Protocol, StreamSocketService>::connect(e, ec);
```

Returns: `this` if `ec` indicates that a connection was successfully established, a null pointer otherwise.

```
template<class T1, class T2, ..., class TN>
  basic_socket_streambuf<Protocol, StreamSocketService>*
    connect(T1 t1, T2 t2, ..., TN tN);
```

Requires: The `Protocol` type must implement the additional requirements for an internet protocol.

Effects: Resets the `streambuf` get and put areas, then opens a connection as if by calling:

```
typename Protocol::resolver::query query(t1, t2, ..., tN);
typename Protocol::resolver resolver(
    this->basic_socket<Protocol, StreamSocketService>::io_service());
typename Protocol::resolver::iterator i = resolver.resolve(query);
typename Protocol::resolver::iterator end;
error_code ec = error::host_not_found;
while (ec && i != end)
{
  this->basic_socket<Protocol, StreamSocketService>::close(ec);
  this->basic_socket<Protocol, StreamSocketService>::connect(*i, ec);
  ++i;
```

```
}
```

Returns: `this` if `ec` indicates that a connection was successfully established, a null pointer otherwise.

```
basic_socket_streambuf<Protocol, StreamSocketService>* close();
```

Effects: If a put area exists, calls `overflow(traits_type::eof())` to flush characters. Calls:

```
error_code ec;
this->basic_socket<Protocol, StreamSocketService>::close(ec);
```

then resets the get and put areas. If the call to `overflow` fails or if the value of `ec` indicates failure then `close` fails.

Returns: `this` on success, a null pointer otherwise.

5.8.1.3. `basic_socket_streambuf` overridden virtual functions

```
virtual int_type underflow();
```

Effects: Behaves according to the description of `basic_streambuf<char>::underflow()`, with the specialisation that a sequence of characters is read from the input sequence as if by reading from the socket into an internal buffer by calling `this->service.receive(this->implementation, ...)`.

Effects: Returns `traits_type::eof()` to indicate failure. Otherwise returns `traits_type::to_int_type(*gptr())`.

```
virtual int_type pbackfail(int_type c = traits_type::eof());
```

Effects: Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

— If `traits_type::eq_int_type(c,traits_type::eof())` returns `false`, and if the function makes a putback position available, and if `traits_type::eq(traits_type::to_char_type(c),gptr()[-1])` returns `true`, decrements the next pointer for the input sequence, `gptr()`.
Returns: `c`.

— If `traits_type::eq_int_type(c,traits_type::eof())` returns `false`, and if the function makes a putback position available, and if the function is permitted to assign to the putback position, decrements the next pointer for the input sequence, and stores `c` there.
Returns: `c`.

— If `traits_type::eq_int_type(c,traits_type::eof())` returns `true`, and if either the input sequence has a putback position available or the function makes a putback position available, decrements the next pointer for the input sequence, `gptr()`.
Returns: `traits_type::not_eof(c)`.

Returns: `traits_type::eof()` to indicate failure.

Notes: The function does not put back a character directly to the input sequence. If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

```
virtual int_type overflow(int_type c = traits_type::eof());
```

Effects: Behaves according to the description of `basic_streambuf<char>::overflow(c)`, except that the behaviour of "consuming characters" is performed by output of the characters to the socket as if by one or more calls to `this->service.send(this->implementation, ...)`.

Returns: `traits_type::not_eof(c)` to indicate success, and `traits_type::eof()` to indicate failure.

```
virtual int sync();
```

Effects: If a put area exists, calls `overflow(traits_type::eof())` to flush characters.

```
virtual streambuf* setbuf(char_type* s, streamsize n);
```

Effects: If `setbuf(0,0)` is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined. "Unbuffered" means that `pbase()` and `pptr()` always return null and output to the socket should appear as soon as possible.

5.8.2. Class template `basic_socket_iostream`

```
namespace std {
  namespace tr2 {
    namespace sys {

      template<class Protocol, class StreamSocketService>
      class basic_socket_iostream :
        public basic_iostream<char>
      {
      public:
        // constructors:
        basic_socket_iostream();
        template<class T1, class T2, ..., class TN>
          explicit basic_socket_iostream(T1 t1, T2 t2, ..., TN tN);

        // members:
        template<class T1, class T2, ..., class TN>
          void connect(T1 t1, T2 t2, ..., TN tN);

        void close();

        basic_socket_streambuf<Protocol, StreamSocketService>* rdbuf() const;

      private:
//      basic_socket_streambuf<Protocol,
//        StreamSocketService> sb;        exposition only
      };

    } // namespace sys
  } // namespace tr2
} // namespace std
```

The class template `basic_socket_iostream<Protocol, StreamSocketService>` supports reading and writing from sockets. It uses a `basic_socket_streambuf<Protocol, StreamSocketService>` object to control the associated sequences. For the sake of exposition, the maintained data is presented here as:

— `sb`, the `basic_socket_streambuf` object.

5.8.2.1. `basic_socket_iostream` constructors

```
basic_socket_iostream();
```

Effects: Constructs an object of class `basic_socket_iostream<Protocol, StreamSocketService>`, initialising the base class with `basic_iostream<char>(&sb)` and initialising `sb` with `basic_socket_streambuf<Protocol, StreamSocketService>()`.

```
template<class T1, class T2, ..., class TN>
  explicit basic_socket_iostream(T1 t1, T2 t2, ..., TN tN);
```

Effects: Constructs an object of class `basic_socket_iostream<Protocol, StreamSocketService>`, initialising the base class with `basic_iostream<char>(&sb)` and initialising `sb` with `basic_socket_streambuf<Protocol, StreamSocketService>()`. Then calls `rdbuf()->connect(t1, t2, ..., tN)`. If that function returns a null pointer, calls `setstate(failbit)`.

5.8.2.2. `basic_socket_iostream` members

```
template<class T1, class T2, ..., class TN>
  void connect(T1 t1, T2 t2, ..., TN tN);
```

Effects: Calls `rdbuf()->connect(t1, t2, ..., tN)`. If that function returns a null pointer, calls `setstate(failbit)` (which may throw `ios_base::failure`).

```
void close();
```

Effects: Calls `rdbuf()->close()`. If that function returns a null pointer, calls `setstate(failbit)` (which may throw `ios_base::failure`).

```
basic_socket_streambuf<Protocol, StreamSocketService>* rdbuf() const;
```

Returns: `const_cast<basic_socket_streambuf<Protocol, StreamSocketService>*>(&sb)`.

5.9. Internet protocol

5.9.1. Requirements

5.9.1.1. Internet protocol requirements

An internet protocol must meet the requirements for a protocol as well as the additional requirements listed below.

In the table below, X denotes an internet protocol class, a denotes a value of type X, and b denotes a value of type X.

Table--InternetProtocol requirements

expression	return type	assertion/note pre/post-conditions
X::resolver	ip::basic_resolver<X>	The type of a resolver for the protocol.
X::v4()	X	Returns an object representing the IP version 4 protocol.
X::v6()	X	Returns an object representing the IP version 6 protocol.
a == b	convertible to bool	Returns whether two protocol objects are equal.
a != b	convertible to bool	Returns !(a == b).

5.9.1.2. Resolver service requirements

A resolver service must meet the requirements for an I/O object service, as well as the additional requirements listed below.

In the table below, X denotes a resolver service class for protocol InternetProtocol, a denotes a value of type X, b denotes a value of type X::implementation_type, q denotes a value of type ip::basic_resolver_query<InternetProtocol>, e denotes a value of type ip::basic_endpoint<InternetProtocol>, ec denotes a value of type error_code, and h denotes a value meeting ResolveHandler requirements.

Table--ResolverService requirements

expression	return type	assertion/note pre/post-condition
a.destroy(b);		From IoObjectService requirements. Implicitly cancels asynchronous resolve operations, as if by calling a.cancel(b, ec).
a.cancel(b, ec);	error_code	Causes any outstanding asynchronous resolve operations to complete as soon as possible. Handlers for cancelled operations shall be passed the error code error::operation_aborted.
a.resolve(b, q, ec);	ip::basic_resolver_iterator< InternetProtocol>	On success, returns an iterator i such that i != ip::basic_resolver_iterator<InternetProtocol>(). Otherwise returns ip::basic_resolver_iterator<InternetProtocol>().
a.async_resolve(b, q, h);		Initiates an asynchronous resolve operation that is performed via the io_service object a.io_service() and behaves according to asynchronous operation requirements. If the operation completes successfully, the ResolveHandler object h shall be invoked with an iterator object i such that the condition i != ip::basic_resolver_iterator<InternetProtocol>() holds. Otherwise it is

expression	return type	assertion/note pre/post-condition
		invoked with `ip::basic_resolver_iterator<InternetProtocol>()`.
`a.resolve(b, e, ec);`	`ip::basic_resolver_iterator< InternetProtocol>`	On success, returns an iterator `i` such that `i != ip::basic_resolver_iterator<InternetProtocol>()`. Otherwise returns `ip::basic_resolver_iterator<InternetProtocol>()`.
`a.async_resolve(b, e, h);`		Initiates an asynchronous resolve operation that is performed via the `io_service` object `a.io_service()` and behaves according to asynchronous operation requirements. If the operation completes successfully, the `ResolveHandler` object `h` shall be invoked with an iterator object `i` such that the condition `i != ip::basic_resolver_iterator<InternetProtocol>()` holds. Otherwise it is invoked with `ip::basic_resolver_iterator<InternetProtocol>()`.

5.9.1.3. Resolve handler requirements

A resolve handler must meet the requirements for a handler. A value `h` of a resolve handler class should work correctly in the expression `h(ec, i)`, where `ec` is an lvalue of type `const error_code` and `s` is an lvalue of type `const ip::basic_resolver_iterator<InternetProtocol>`. `InternetProtocol` is the template parameter of the `resolver_service` which is used to initiate the asynchronous operation.

5.9.2. Class `ip::address`

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        class address
        {
        public:
          // constructors:
          address();
          address(const address_v4& addr);
          address(const address_v6& addr);

          // assignment:
          address& operator=(const address_v4& addr);
          address& operator=(const address_v6& addr);

          // members:
          bool is_multicast() const;
          bool is_v4() const;
          bool is_v6() const;
          address_v4 to_v4() const;
          address_v6 to_v6() const;
          string to_string() const;
          string to_string(error_code& ec) const;

          // static members:
          static address from_string(const string& str);
          static address from_string(const string& str, error_code& ec);
        };

        // address comparisons:
```

```
        bool operator==(const address& a, const address& b);
        bool operator!=(const address& a, const address& b);
        bool operator< (const address& a, const address& b);
        bool operator> (const address& a, const address& b);
        bool operator<=(const address& a, const address& b);
        bool operator>=(const address& a, const address& b);

        // address I/O:
        template<class CharT, class Traits>
          basic_ostream<CharT, Traits>& operator<<(
            basic_ostream<CharT, Traits>& os, const address& addr);

      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.9.2.1. `ip::address` constructors

`address();`

> *Effects:* Constructs an object of class `address`.
>
> *Postconditions:* The postconditions of this function are indicated in the table below.

Table--address::address() effects

expression	value
`is_v4()`	`true`
`is_v6()`	`false`
`to_v4()`	`address_v4()`

`address(const address_v4& addr);`

> *Effects:* Constructs an object of class `address`, initialising it with the specified IP version 4 address.
>
> *Postconditions:* The postconditions of this function are indicated in the table below.

Table--address::address(const address_v4&) effects

expression	value
`is_v4()`	`true`
`is_v6()`	`false`
`to_v4()`	`addr`

`address(const address_v6& addr);`

> *Effects:* Constructs an object of class `address`, initialising it with the specified IP version 6 address.
>
> *Postconditions:* The postconditions of this function are indicated in the table below.

Table--address::address(const address_v6&) effects

expression	value
`is_v4()`	`false`
`is_v6()`	`true`
`to_v6()`	`addr`

5.9.2.2. `ip::address` assignment

`address& operator=(const address_v4& addr);`

> *Effects:* Assigns the specified IP version 4 address into an object of class `address`.
>
> *Returns:* `*this`.
>
> *Postconditions:* The postconditions of this function are indicated in the table below.

Table--address& address::operator=(const address_v4&) effects

expression	value
`is_v4()`	`true`
`is_v6()`	`false`
`to_v4()`	`addr`

```
address& operator=(const address_v6& addr);
```

Effects: Assigns the specified IP version 6 address into an object of class `address`.

Returns: `*this`.

Postconditions: The postconditions of this function are indicated in the table below.

Table--address& address::operator=(const address_v6&) effects

expression	value
`is_v4()`	`true`
`is_v6()`	`false`
`to_v6()`	`addr`

5.9.2.3. `ip::address` members

```
bool is_multicast() const;
```

Returns: `is_v4() ? to_v4().is_multicast() : to_v6().is_multicast()`.

```
bool is_v4() const;
```

Returns: `true` if the object contains an IP version 4 address.

```
bool is_v6() const;
```

Returns: `true` if the object contains an IP version 6 address.

```
address_v4 to_v4() const;
```

Returns: The IP version 4 address contained by the object.

Throws: `bad_cast` if `is_v4()` is false.

```
address_v6 to_v6() const;
```

Returns: The IP version 6 address contained by the object.

Throws: `bad_cast` if `is_v6()` is false.

```
string to_string() const;
string to_string(error_code& ec) const;
```

Returns: `is_v4() ? to_v4().to_string(ec) : to_v6().to_string(ec)`.

5.9.2.4. `ip::address` static members

```
static address from_string(const string& str);
static address from_string(const string& str, error_code& ec);
```

Effects: Converts a string representation of an address into an object of class `address`, as if by calling:

```
address a;
address_v6 v6a = address_v6::from_string(str, ec);
if (!ec)
  a = v6a;
else
{
  address_v4 v4a = address_v4::from_string(str, ec);
  if (!ec)
    a = v4a;
}
```

Returns: `a`.

5.9.2.5. `ip::address` comparisons

```
bool operator==(const address& a, const address& b);
```

Returns: `(a.is_v4() && b.is_v4() && a.to_v4() == b.to_v4()) || (a.is_v6() && b.is_v6() && a.to_v6() == b.to_v6())`.

```
bool operator!=(const address& a, const address& b);
```

Returns: `!(a == b)`.

```
bool operator< (const address& a, const address& b);
```

Returns: `(a.is_v4() && b.is_v4() && a.to_v4() < b.to_v4()) || (a.is_v6() && b.is_v6() && a.to_v6() < b.to_v6()) || (a.is_v4() && b.is_v6())`.

```
bool operator> (const address& a, const address& b);
```

Returns: `b < a`.

```
bool operator<=(const address& a, const address& b);
```

Returns: `!(b < a)`.

```
bool operator>=(const address& a, const address& b);
```

Returns: `!(a < b)`.

5.9.2.6. `ip::address` I/O

```
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os, const address& addr);
```

Effects: Outputs the string representation of the address to the stream, as if it were implemented as follows:

```
error_code ec;
string s = addr.to_string(ec);
if (ec)
  os.setstate(ios_base::failbit); // may throw ios::failure
else
  for (string::iterator i = s.begin(); i != s.end(); ++i)
    os << os.widen(*i);
```

Returns: `os`.

5.9.3. Class `ip::address_v4`

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        class address_v4
        {
        public:
          // types:
          typedef array<unsigned char, 4> bytes_type;

          // constructors:
          address_v4();
          explicit address_v4(const bytes_type& bytes);
          explicit address_v4(unsigned long val);

          // members:
          bool is_class_a() const;
          bool is_class_b() const;
          bool is_class_c() const;
          bool is_multicast() const;
          bytes_type to_bytes() const;
          unsigned long to_ulong() const;
          string to_string() const;
```

```
        string to_string(error_code& ec) const;

        // static members:
        static address_v4 from_string(const string& str);
        static address_v4 from_string(const string& str, error_code& ec);
        static address_v4 any();
        static address_v4 loopback();
        static address_v4 broadcast();
        static address_v4 broadcast(const address_v4& addr,
          const address_v4& mask);
        static address_v4 netmask(const address_v4& addr);
      };

      // address_v4 comparisons:
      bool operator==(const address_v4& a, const address_v4& b);
      bool operator!=(const address_v4& a, const address_v4& b);
      bool operator< (const address_v4& a, const address_v4& b);
      bool operator> (const address_v4& a, const address_v4& b);
      bool operator<=(const address_v4& a, const address_v4& b);
      bool operator>=(const address_v4& a, const address_v4& b);

      // address_v4 I/O:
      template<class CharT, class Traits>
        basic_ostream<CharT, Traits>& operator<<(
          basic_ostream<CharT, Traits>& os, const address_v4& addr);

    } // namespace ip
  } // namespace sys
 } // namespace tr2
} // namespace std
```

5.9.3.1. `ip::address_v4` constructors

`address_v4();`

Effects: Constructs an object of class `address_v4`.

Postconditions: The postconditions of this function are indicated in the table below.

Table--address_v4::address_v4() effects

expression	value
`to_bytes()`	`{0, 0, 0, 0}`
`to_ulong()`	`0`

`address_v4(const bytes_type& bytes);`

Effects: Constructs an object of class `address_v4`.

Requires: Each element of the array `bytes` is in the range `[0, 0xFF]`.

Throws: `out_of_range` if any element of the array `bytes` is not in the range `[0, 0xFF]`. [*Note:* For implementations where `UCHAR_MAX == 0xFF`, no out-of-range detection is needed. *—end note*]

Postconditions: The postconditions of this function are indicated in the table below.

Table--address_v4::address_v4(const bytes_type&) effects

expression	value
`to_bytes()`	`{ bytes[0], bytes[1], bytes[2], bytes[3] }`
`to_ulong()`	`(bytes[0] << 24) | (bytes[1] << 16) | (bytes[2] << 8) | bytes[3]`

`address_v4(unsigned long val);`

Effects: Constructs an object of class `address_v4`.

Requires: `val` is in the range `[0, 0xFFFFFFFF]`.

Throws: `out_of_range` if `val` is not in the range `[0, 0xFFFFFFFF]`. [*Note:* For implementations where `ULONG_MAX == 0xFFFFFFFF`, no out-of-range detection is needed. —*end note*]

Postconditions: The postconditions of this function are indicated in the table below.

Table--address_v4::address_v4(unsigned long) effects

expression	value
`to_bytes()`	`{ (val >> 24) & 0xFF, (val >> 16) & 0xFF, (val >> 8) & 0xFF, val & 0xFF }`
`to_ulong()`	`val`

5.9.3.2. `ip::address_v4` members

```
bool is_class_a() const;
```

Returns: `(to_ulong() & 0x80000000) == 0`.

```
bool is_class_b() const;
```

Returns: `(to_ulong() & 0xC0000000) == 0x80000000`.

```
bool is_class_c() const;
```

Returns: `(to_ulong() & 0xE0000000) == 0xC0000000`.

```
bool is_multicast() const;
```

Returns: `(to_ulong() & 0xF0000000) == 0xE0000000`.

```
bytes_type to_bytes() const;
```

Returns: A representation of the address in network byte order.

```
unsigned long to_ulong() const;
```

Returns: A representation of the address in host byte order.

```
string to_string() const;
string to_string(error_code& ec) const;
```

Effects: Converts an address into a string representation, as if by *POSIX* `inet_ntop()` when invoked with address family `AF_INET`.

Returns: If successful, the string representation of the address. Otherwise `string()`.

5.9.3.3. `ip::address_v4` static members

```
static address_v4 from_string(const string& str);
static address_v4 from_string(const string& str, error_code& ec);
```

Effects: Converts a string representation of an address into a corresponding `address_v4` value, as if by *POSIX* `inet_pton()` when invoked with address family `AF_INET`.

Returns: If successful, an `address_v4` value corresponding to the string `str`. Otherwise `address_v4()`.

```
static address_v4 any();
```

Returns: `address_v4()`.

```
static address_v4 loopback();
```

Returns: `address_v4(0x7F000001)`.

```
static address_v4 broadcast();
```

Returns: `address_v4(0xFFFFFFFF)`.

```
static address_v4 broadcast(const address_v4& addr, const address_v4& mask);
```

Returns: `address_v4(addr.to_ulong() | ~mask.to_ulong())`.

```
static address_v4 netmask(const address_v4& addr);
```

If `addr.is_class_a()` is true, returns `address_v4(0xFF000000)`. If `addr.is_class_b()` is true, returns `address_v4(0xFFFF0000)`. If `addr.is_class_c()` is true, returns `address_v4(0xFFFFFF00)`. Otherwise returns `address_v4(0xFFFFFFFF)`.

5.9.3.4. `ip::address_v4` comparisons

```
bool operator==(const address_v4& a, const address_v4& b);
```

Returns: `a.to_ulong() == b.to_ulong()`.

```
bool operator!=(const address_v4& a, const address_v4& b);
```

Returns: `a.to_ulong() != b.to_ulong()`.

```
bool operator< (const address_v4& a, const address_v4& b);
```

Returns: `a.to_ulong() < b.to_ulong()`.

```
bool operator> (const address_v4& a, const address_v4& b);
```

Returns: `a.to_ulong() > b.to_ulong()`.

```
bool operator<=(const address_v4& a, const address_v4& b);
```

Returns: `a.to_ulong() <= b.to_ulong()`.

```
bool operator>=(const address_v4& a, const address_v4& b);
```

Returns: `a.to_ulong() >= b.to_ulong()`.

5.9.3.5. `ip::address_v4` I/O

```
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os, const address_v4& addr);
```

Effects: Outputs the string representation of the address to the stream, as if it were implemented as follows:

```
error_code ec;
string s = addr.to_string(ec);
if (ec)
  os.setstate(ios_base::failbit); // may throw ios::failure
else
  for (string::iterator i = s.begin(); i != s.end(); ++i)
    os << os.widen(*i);
```

Returns: `os`.

5.9.4. Class `ip::address_v6`

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        class address_v6
        {
        public:
          // types:
          typedef array<unsigned char, 16> bytes_type;

          // constructors:
          address_v6();
          explicit address_v6(const bytes_type& bytes,
                              unsigned long scope_id = 0);

          // members:
          void scope_id(unsigned long id);
          unsigned long scope_id() const;
          bool is_unspecified() const;
          bool is_loopback() const;
          bool is_multicast() const;
```

```
      bool is_link_local() const;
      bool is_site_local() const;
      bool is_v4_mapped() const;
      bool is_v4_compatible() const;
      bool is_multicast_node_local() const;
      bool is_multicast_link_local() const;
      bool is_multicast_site_local() const;
      bool is_multicast_org_local() const;
      bool is_multicast_global() const;
      bytes_type to_bytes() const;
      string to_string() const;
      string to_string(error_code& ec) const;
      address_v4 to_v4() const;

      // static members:
      static address_v6 from_string(const string& str);
      static address_v6 from_string(const string& str, error_code& ec);
      static address_v6 any();
      static address_v6 loopback();
      static address_v6 v4_mapped(const address_v4& addr);
      static address_v6 v4_compatible(const address_v4& addr);
    };

    // address_v6 comparisons:
    bool operator==(const address_v6& a, const address_v6& b);
    bool operator!=(const address_v6& a, const address_v6& b);
    bool operator< (const address_v6& a, const address_v6& b);
    bool operator> (const address_v6& a, const address_v6& b);
    bool operator<=(const address_v6& a, const address_v6& b);
    bool operator>=(const address_v6& a, const address_v6& b);

    // address_v6 I/O:
    template<class CharT, class Traits>
      basic_ostream<CharT, Traits>& operator<<(
        basic_ostream<CharT, Traits>& os, const address_v6& addr);

  } // namespace ip
 } // namespace sys
} // namespace tr2
} // namespace std
```

[*Note:* The implementations of the functions `is_unspecified`, `is_loopback`, `is_multicast`, `is_link_local`, `is_site_local`, `is_v4_mapped`, `is_v4_compatible`, `is_multicast_node_local`, `is_multicast_link_local`, `is_multicast_site_local`, `is_multicast_org_local` and `is_multicast_global` are determined by [RFC3513]. — *end note*]

5.9.4.1. `ip::address_v6` constructors

`address_v6();`

Effects: Constructs an object of class `address_v6`.

Postconditions: The postconditions of this function are indicated in the table below.

Table--address_v6::address_v6() effects

expression	value
`is_unspecified()`	`true`
`scope_id()`	`0`

```
explicit address_v6(const bytes_type& bytes,
                    unsigned long scope_id = 0);
```

Effects: Constructs an object of class `address_v6`.

Postconditions: The postconditions of this function are indicated in the table below.

Table--address_v6::address_v6() effects

expression	value
`to_bytes()`	`bytes`
`scope_id()`	`scope_id`

5.9.4.2. `ip::address_v6` members

```
void scope_id(unsigned long id);
```

Postconditions: `scope_id() == id`.

```
unsigned long scope_id() const;
```

Returns: The scope identifier associated with the address.

```
bool is_unspecified() const;
```

Returns: A boolean indicating whether the `address_v6` object represents an unspecified address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[ 0] == 0 && b[ 1] == 0 && b[ 2] == 0 && b[ 3] == 0
    && b[ 4] == 0 && b[ 5] == 0 && b[ 6] == 0 && b[ 7] == 0
    && b[ 8] == 0 && b[ 9] == 0 && b[10] == 0 && b[11] == 0
    && b[12] == 0 && b[13] == 0 && b[14] == 0 && b[15] == 0;
```

```
bool is_loopback() const;
```

Returns: A boolean indicating whether the `address_v6` object represents a loopback address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[ 0] == 0 && b[ 1] == 0 && b[ 2] == 0 && b[ 3] == 0
    && b[ 4] == 0 && b[ 5] == 0 && b[ 6] == 0 && b[ 7] == 0
    && b[ 8] == 0 && b[ 9] == 0 && b[10] == 0 && b[11] == 0
    && b[12] == 0 && b[13] == 0 && b[14] == 0 && b[15] == 1;
```

```
bool is_multicast() const;
```

Returns: A boolean indicating whether the `address_v6` object represents a multicast address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFF;
```

```
bool is_link_local() const;
```

Returns: A boolean indicating whether the `address_v6` object represents a unicast link-local address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFE && (b[1] & 0xC0) == 0x80;
```

```
bool is_site_local() const;
```

Returns: A boolean indicating whether the `address_v6` object represents a unicast site-local address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFE && (b[1] & 0xC0) == 0xC0;
```

```
bool is_v4_mapped() const;
```

Returns: A boolean indicating whether the `address_v6` object represents an IPv4-mapped IPv6 address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[ 0] == 0 && b[ 1] == 0 && b[ 2] == 0 && b[ 3] == 0
    && b[ 4] == 0 && b[ 5] == 0 && b[ 6] == 0 && b[ 7] == 0
    && b[ 8] == 0 && b[ 9] == 0 && b[10] == 0xFF && b[11] == 0xFF;
```

```
bool is_v4_compatible() const;
```

Returns: A boolean indicating whether the `address_v6` object represents an IPv4-compatible IPv6 address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[ 0] == 0 && b[ 1] == 0 && b[ 2] == 0 && b[ 3] == 0
    && b[ 4] == 0 && b[ 5] == 0 && b[ 6] == 0 && b[ 7] == 0
    && b[ 8] == 0 && b[ 9] == 0 && b[10] == 0 && b[11] == 0
    && !is_unspecified() && !is_loopback();
```

```
bool is_multicast_node_local() const;
```

Returns: A boolean indicating whether the `address_v6` object represents a multicast node-local address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFF && (b[1] & 0x0F) == 0x01;
```

```
bool is_multicast_link_local() const;
```

Returns: A boolean indicating whether the `address_v6` object represents a multicast link-local address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFF && (b[1] & 0x0F) == 0x02;
```

```
bool is_multicast_site_local() const;
```

Returns: A boolean indicating whether the `address_v6` object represents a multicast site-local address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFF && (b[1] & 0x0F) == 0x05;
```

```
bool is_multicast_org_local() const;
```

Returns: A boolean indicating whether the `address_v6` object represents a multicast organisation-local address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFF && (b[1] & 0x0F) == 0x08;
```

```
bool is_multicast_global() const;
```

Returns: A boolean indicating whether the `address_v6` object represents a multicast organisation-local address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFF && (b[1] & 0x0F) == 0x0E;
```

```
bytes_type to_bytes() const;
```

Returns: A representation of the address in network byte order.

```
string to_string() const;
string to_string(error_code& ec) const;
```

Effects: Converts an address into a string representation. If `scope_id() == 0`, converts as if by *POSIX* `inet_ntop()` when invoked with address family `AF_INET6`. If `scope_id() != 0`, the format is `address%scope-id`, where `address` is the string representation of the equivalent address having `scope_id() == 0`, and `scope-id` is an implementation-defined string representation of the scope identifier.

Returns: If successful, the string representation of the address. Otherwise `string()`.

```
address_v4 to_v4() const;
```

Requires: `is_v4_mapped() || is_v4_compatible()`.

Returns: An `address_v4` object corresponding to the IPv4-mapped or IPv4 compatible IPv6 address, as if computed by the following method:

```
bytes_type v6b = to_bytes();
address_v4::bytes_type v4b = { v6b[12], v6b[13], v6b[14], v6b[15] };
return address_v4(v4b);
```

Throws: `bad_cast` if `is_v4_mapped()` is `false` and `is_v4_compatible()` is `false`.

5.9.4.3. `ip::address_v6` static members

```
static address_v6 from_string(const string& str);
static address_v6 from_string(const string& str, error_code& ec);
```

Effects: Converts a string representation of an address into a corresponding `address_v6` value. The format is either `address` or `address%scope-id`, where `address` is in the format specified by *POSIX* `inet_pton()`

when invoked with address family `AF_INET6`, and `scope-id` is an optional string specifying the scope identifier. All implementations shall accept as `scope-id` a string representation of an unsigned decimal integer. It is implementation-defined whether alternative scope identifier representations are permitted. If `scope-id` is not supplied, an `address_v6` object shall be returned such that `scope_id() == 0`.

Returns: If successful, an `address_v6` value corresponding to the string `str`. Otherwise returns `address_v6()`.

```
static address_v6 any();
```

Returns: `address_v6()`.

```
static address_v6 loopback();
```

Returns: An address `a` such that the condition `a.is_loopback()` holds.

```
static address_v6 v4_mapped(const address_v4& addr);
```

Returns: An `address_v6` object containing the IPv4-mapped IPv6 address corresponding to the specified IPv4 address, as if computed by the following method:

```
address_v4::bytes_type v4b = addr.to_bytes();
bytes_type v6b = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                   0xFF, 0xFF, v4b[0], v4b[1], v4b[2], v4b[3] };
return address_v6(v6b);
```

```
static address_v6 v4_compatible(const address_v4& addr);
```

Returns: An `address_v6` object containing the IPv4-compatible IPv6 address corresponding to the specified IPv4 address, as if computed by the following method:

```
address_v4::bytes_type v4b = addr.to_bytes();
bytes_type v6b = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                   v4b[0], v4b[1], v4b[2], v4b[3] };
return address_v6(v6b);
```

5.9.4.4. `ip::address_v6` comparisons

```
bool operator==(const address_v6& a, const address_v6& b);
```

Returns: `a.to_bytes() == b.to_bytes() && a.scope_id() == b.scope_id()`.

```
bool operator!=(const address_v6& a, const address_v6& b);
```

Returns: `!(a == b)`.

```
bool operator< (const address_v6& a, const address_v6& b);
```

Returns: `a.to_bytes() < b.to_bytes() || (!(b.to_bytes() < a.to_bytes()) && a.scope_id() < b.scope_id())`.

```
bool operator> (const address_v6& a, const address_v6& b);
```

Returns: `b < a`.

```
bool operator<=(const address_v6& a, const address_v6& b);
```

Returns: `!(b < a)`.

```
bool operator>=(const address_v6& a, const address_v6& b);
```

Returns: `!(a < b)`.

5.9.4.5. `ip::address_v6` I/O

```
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os, const address_v6& addr);
```

Effects: Outputs the string representation of the address to the stream, as if it were implemented as follows:

```
error_code ec;
string s = addr.to_string(ec);
if (ec)
  os.setstate(ios_base::failbit); // may throw ios::failure
```

```
else
  for (string::iterator i = s.begin(); i != s.end(); ++i)
    os << os.widen(*i);
```

Returns: `os`.

5.9.5. Class template `ip::basic_endpoint`

Instances of the `basic_endpoint` class template meet the requirements for an [`Endpoint`].

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        template<class InternetProtocol>
        class basic_endpoint
        {
        public:
          // types:
          typedef InternetProtocol protocol_type;

          // constructors:
          basic_endpoint();
          basic_endpoint(const InternetProtocol& proto,
                         unsigned short port_num);
          basic_endpoint(const ip::address& addr,
                         unsigned short port_num);

          // members:
          InternetProtocol protocol() const;
          ip::address address() const;
          void address(const ip::address& addr);
          unsigned short port() const;
          void port(unsigned short port_num);
        };

        // basic_endpoint comparisons:
        template<class InternetProtocol>
          bool operator==(const basic_endpoint<InternetProtocol>& a,
                          const basic_endpoint<InternetProtocol>& b);
        template<class InternetProtocol>
          bool operator!=(const basic_endpoint<InternetProtocol>& a,
                          const basic_endpoint<InternetProtocol>& b);
        template<class InternetProtocol>
          bool operator< (const basic_endpoint<InternetProtocol>& a,
                          const basic_endpoint<InternetProtocol>& b);
        template<class InternetProtocol>
          bool operator> (const basic_endpoint<InternetProtocol>& a,
                          const basic_endpoint<InternetProtocol>& b);
        template<class InternetProtocol>
          bool operator<=(const basic_endpoint<InternetProtocol>& a,
                          const basic_endpoint<InternetProtocol>& b);
        template<class InternetProtocol>
          bool operator>=(const basic_endpoint<InternetProtocol>& a,
                          const basic_endpoint<InternetProtocol>& b);

        // basic_endpoint I/O:
        template<class CharT, class Traits, class InternetProtocol>
          basic_ostream<CharT, Traits>& operator<<(
            basic_ostream<CharT, Traits>& os,
            const basic_endpoint<InternetProtocol>& ep);

      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

Extensible implementations shall provide the following member functions:

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        template<class InternetProtocol>
        class basic_endpoint
```

```
        {
        public:
          unspecified* data();
          const unspecified* data() const;
          size_t size() const;
          void resize(size_t s);
          size_t capacity() const;
          // remainder unchanged
        private:
//        sockaddr_storage data_;  exposition only
        };

      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.9.5.1. `ip::basic_endpoint` constructors

```
basic_endpoint();
```

Effects: Constructs an object of class `basic_endpoint<InternetProtocol>`.

Postconditions: The postconditions of this function are indicated in the table below.

Table--basic_endpoint<InternetProtocol>::basic_endpoint() effects

expression	value
`address()`	`ip::address(ip::address_v4())`
`port()`	`0`

```
basic_endpoint(const InternetProtocol& proto,
               unsigned short port_num);
```

Effects: Constructs an object of class `basic_endpoint<InternetProtocol>` with the specified protocol and port number.

Postconditions: The postconditions of this function are indicated in the table below.

Table--basic_endpoint<InternetProtocol>::basic_endpoint(const InternetProtocol&, unsigned short) effects

expression	value
`address()`	`ip::address(ip::address_v6())` if `proto == Protocol::v6()`, otherwise `ip::address(ip::address_v4())`
`port()`	`port_num`

```
basic_endpoint(const ip::address& addr,
               unsigned short port_num);
```

Effects: Constructs an object of class `basic_endpoint<InternetProtocol>` with the specified address and port number.

Postconditions: The postconditions of this function are indicated in the table below.

Table--basic_endpoint<InternetProtocol>::basic_endpoint(const ip::address&, unsigned short) effects

expression	value
`address()`	`addr`
`port()`	`port_num`

5.9.5.2. `ip::basic_endpoint` members

```
InternetProtocol protocol() const;
```

Returns: `Protocol::v6()` if the expression `address().is_v6()` is true, otherwise `Protocol::v4()`.

```
ip::address address() const;
```

Returns: The address associated with the endpoint.

```
void address(const ip::address& addr);
```

Effects: Modifies the address associated with the endpoint.

Postconditions: `address() == addr`.

```
unsigned short port() const;
```

Returns: The port number associated with the endpoint.

```
void port(unsigned short port_num);
```

Effects: Modifies the port number associated with the endpoint.

Postconditions: `port() == port_num`.

5.9.5.3. `ip::basic_endpoint` comparisons

```
template<class InternetProtocol>
  bool operator==(const basic_endpoint<InternetProtocol>& a,
                  const basic_endpoint<InternetProtocol>& b);
```

Returns: `a.address() == b.address() && a.port() == b.port())`.

```
template<class InternetProtocol>
  bool operator!=(const basic_endpoint<InternetProtocol>& a,
                  const basic_endpoint<InternetProtocol>& b);
```

Returns: `!(a == b)`.

```
template<class InternetProtocol>
  bool operator< (const basic_endpoint<InternetProtocol>& a,
                  const basic_endpoint<InternetProtocol>& b);
```

Returns: `a.address() < b.address() || (!(b.address() < a.address()) && a.port() < b.port())`.

```
template<class InternetProtocol>
  bool operator> (const basic_endpoint<InternetProtocol>& a,
                  const basic_endpoint<InternetProtocol>& b);
```

Returns: `b < a`.

```
template<class InternetProtocol>
  bool operator<=(const basic_endpoint<InternetProtocol>& a,
                  const basic_endpoint<InternetProtocol>& b);
```

Returns: `!(b < a)`.

```
template<class InternetProtocol>
  bool operator>=(const basic_endpoint<InternetProtocol>& a,
                  const basic_endpoint<InternetProtocol>& b);
```

Returns: `!(a < b)`.

5.9.5.4. `ip::basic_endpoint` I/O

```
template<class CharT, class Traits, class InternetProtocol>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os,
    const basic_endpoint<InternetProtocol>& ep);
```

Effects: Outputs a representation of the endpoint to the stream, as if it were implemented as follows:

```
basic_ostringstream<CharT, Traits> ss;
if (ep.protocol() == Protocol::v6())
  ss << ss.widen('[') << ss.address() << ss.widen(']');
else
  ss << ep.address();
ss << ss.widen(':') << ep.port();
os << ss.str();
```

Returns: `os`.

[*Note:* The representation of the endpoint when it contains an IP version 6 address is based on [RFC2732]. —*end note*]

5.9.5.5. `ip::basic_endpoint` members (extensible implementations)

`unspecified* data();`

Returns: `&data_`.

`const unspecified* data() const;`

Returns: `&data_`.

`size_t size() const;`

Returns: `sizeof(sockaddr_in6)` if `protocol().family() == AF_INET`, otherwise `sizeof(sockaddr_in)`.

`void resize(size_t s);`

Throws: `length_error` if the condition `protocol().family() == AF_INET6 && s != sizeof(sockaddr_in6)` || `protocol().family() != AF_INET6 && s != sizeof(sockaddr_in)` is true.

`size_t capacity() const;`

Returns: `sizeof(data_)`.

5.9.6. Class `ip::resolver_query_base`

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        class resolver_query_base
        {
        public:
          typedef T1 flags;
          static const flags passive;
          static const flags canonical_name;
          static const flags numeric_host;
          static const flags numeric_service;
          static const flags v4_mapped;
          static const flags all_matching;
          static const flags address_configured;

        protected:
          resolver_query_base();
          ~resolver_query_base();
        };

      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

`resolver_query_base` defines a bitmask type, `flags`. The flags constants have bitwise-distinct values. The meanings and *POSIX* equivalents for each flag are defined in the table below.

Table--resolver flags

flag	meaning	*POSIX* equivalent
`passive`	Returned endpoints are intended for use as locally bound socket endpoints.	`AI_PASSIVE`
`canonical_name`	Determine the canonical name of the host specified in the query.	`AI_CANONNAME`
`numeric_host`	Host name should be treated as a numeric string defining an IPv4 or IPv6 address and no host name resolution should be attempted.	`AI_NUMERICHOST`
`numeric_service`	Service name should be treated as a numeric string defining a port number and no service name resolution should be attempted.	`AI_NUMERICSERV`
`v4_mapped`	If the query protocol is specified as an IPv6 protocol, return IPv4-mapped IPv6 addresses on finding no IPv6 addresses.	`AI_V4MAPPED`
`all_matching`	If used with `v4_mapped`, return all matching IPv6 and IPv4 addresses.	`AI_ALL`

flag	meaning	*POSIX* equivalent
`address_configured`	Only return IPv4 addresses if a non-loopback IPv4 address is configured for the system. Only return IPv6 addresses if a non-loopback IPv6 address is configured for the system.	`AI_ADDRCONFIG`

5.9.7. Class template `ip::basic_resolver_entry`

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        template<class InternetProtocol>
        class basic_resolver_entry
        {
        public:
          // types:
          typedef InternetProtocol protocol_type;
          typedef typename InternetProtocol::endpoint endpoint_type;

          // constructors:
          basic_resolver_entry();
          basic_resolver_entry(const endpoint_type& ep, const string& h,
                               const string& s);

          // members:
          endpoint_type endpoint() const;
          operator endpoint_type() const;
          string host_name() const;
          string service_name() const;
        };

      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.9.7.1. `ip::basic_resolver_entry` constructors

```
basic_resolver_entry();
```

Effects: Constructs an object of class `basic_resolver_entry<InternetProtocol>`.

Postconditions: The postconditions of this function are indicated in the table below.

Table--basic_resolver_entry<InternetProtocol>::basic_resolver_entry() effects

expression	value
`endpoint()`	`basic_endpoint<InternetProtocol>()`
`host_name()`	`string()`
`service_name()`	`string()`

```
basic_resolver_entry(const endpoint_type& ep, const string& h,
                     const string& s);
```

Effects: Constructs an object of class `basic_resolver_entry<InternetProtocol>` with the specified endpoint, host name and service name.

Postconditions: The postconditions of this function are indicated in the table below.

Table--basic_resolver_entry<InternetProtocol>::basic_resolver_entry() effects

expression	value
`endpoint()`	`ep`
`host_name()`	`h`
`service_name()`	`s`

5.9.7.2. `ip::basic_resolver_entry` members

```
endpoint_type endpoint() const;
```

Returns: The endpoint associated with the resolver entry.

```
operator endpoint_type() const;
```

Returns: `endpoint()`.

```
string host_name() const;
```

Returns: The host name associated with the resolver entry.

```
string service_name() const;
```

Returns: The service name associated with the resolver entry.

5.9.8. Class template `ip::basic_resolver_iterator`

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        template<class InternetProtocol>
        class basic_resolver_iterator :
          public iterator<forward_iterator_tag,
            const basic_resolver_entry<InternetProtocol> >
        {
        public:
          // types:
          typedef InternetProtocol protocol_type;
          typedef typename InternetProtocol::endpoint endpoint_type;

          // constructors:
          basic_resolver_iterator();

          // other members as required by
          //   C++ Std, 24.1.3 Forward iterators [lib.forward.iterators]
        };

      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.9.8.1. `ip::basic_resolver_iterator` constructors

```
basic_resolver_iterator();
```

Effects: Initialises an object of class `basic_resolver_iterator<InternetProtocol>` so that it represents an end iterator.

5.9.9. Class template `ip::basic_resolver_query`

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        template<class InternetProtocol>
        class basic_resolver_query :
          public resolver_query_base
        {
        public:
          // types:
          typedef InternetProtocol protocol_type;

          // constructors:
          basic_resolver_query();
          basic_resolver_query(const string& service_name,
                               flags f = passive | address_configured);
          basic_resolver_query(const InternetProtocol& proto,
                               const string& service_name,
                               flags f = passive | address_configured);
          basic_resolver_query(const string& host_name,
                               const string& service_name,
                               flags f = address_configured);
```

```
          basic_resolver_query(const InternetProtocol& proto,
                               const string& host_name,
                               const string& service_name,
                               flags f = address_configured);
        };

      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

The `basic_resolver_query` class encapsulates values used in name resolution. The meanings of the constructor arguments are defined below as if the name resolution is performed using *POSIX* `getaddrinfo()`:

— *proto* is used to populate the `ai_family`, `ai_socktype` and `ai_protocol` fields of the `addrinfo` structure passed as the *hints* argument to *POSIX* `getaddrinfo()`.

— *host_name* is passed as the *nodename* argument to *POSIX* `getaddrinfo()`.

— *service_name* is passed as the *servname* argument to *POSIX* `getaddrinfo()`.

— *flags* is used to populate the `ai_flags` field of the `addrinfo` structure passed as the *hints* argument to *POSIX* `getaddrinfo()`.

If a default-constructed `basic_resolver_query` object is used in a call to `resolver_service<>::resolve()`, the results are undefined.

5.9.10. Class template `ip::resolver_service`

Instances of the `resolver_service` class template meet the requirements of a `ResolverService`.

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        template<class InternetProtocol>
        class resolver_service :
          public io_service::service
        {
        public:
          static io_service::id id;

          // types:
          typedef InternetProtocol protocol_type;
          typedef typename InternetProtocol::endpoint endpoint_type;
          typedef basic_resolver_query<InternetProtocol> query_type;
          typedef basic_resolver_iterator<InternetProtocol> iterator_type;
          typedef unspecified implementation_type;

          // constructors:
          explicit resolver_service(io_service& ios);

          // members:
          void construct(implementation_type& impl);

          void destroy(implementation_type& impl);

          error_code cancel(implementation_type& impl, error_code& ec);

          iterator_type resolve(implementation_type& impl,
                                const query_type& q, error_code& ec);

          template<class ResolveHandler>
            void async_resolve(implementation_type& impl,
                               const query_type& q, ResolveHandler handler);

          iterator_type resolve(implementation_type& impl,
                                const endpoint_type& e, error_code& ec);

          template<class ResolveHandler>
            void async_resolve(implementation_type& impl,
                               const endpoint_type& e,
                               ResolveHandler handler);

        private:
          virtual void shutdown_service();
```

```
    };

   } // namespace ip
  } // namespace sys
 } // namespace tr2
} // namespace std
```

5.9.10.1. `ip::resolver_service` constructors

```
explicit resolver_service(io_service& ios);
```

Effects: Constructs an object of class `resolver_service<InternetProtocol>`, initialising the base class with `io_service::service(ios)`.

5.9.10.2. `ip::resolver_service` members

```
void shutdown_service();
```

Effects: Destroys all copies of user-defined handler objects owned by the service.

```
void construct(implementation_type& impl);
```

Effects: Initialises the resolver implementation `impl`.

```
void destroy(implementation_type& impl);
```

Effects: Cleans up resources owned by the resolver implementation `impl`. Cancels asynchronous operations associated with `impl` as if by performing:

```
error_code ec;
cancel(impl, ec);
```

```
error_code cancel(implementation_type& impl, error_code& ec);
```

Effects: Causes any outstanding asynchronous operations to complete as soon as possible. Handlers for cancelled operations shall be passed the error code `error::operation_aborted`.

Returns: `ec`.

```
iterator_type resolve(implementation_type& impl,
                      const query_type& q, error_code& ec);
```

Effects: Translates a query into a sequence of `basic_resolver_entry<InternetProtocol>` objects, as if by *POSIX* `getaddrinfo()`.

Returns: On success, an iterator object `i` such that the condition `i != iterator_type()` holds. Otherwise `iterator_type()`.

```
template<class ResolveHandler>
  void async_resolve(implementation_type& impl,
                     const query_type& q, ResolveHandler handler);
```

Effects: Initiates an asynchronous operation to translate a query into a sequence of `basic_resolver_entry<InternetProtocol>` objects, as if by *POSIX* `getaddrinfo()`. The operation is performed via the `io_service` object returned by `io_service::service::io_service()` and behaves according to asynchronous operation requirements.

If the operation completes successfully, the `ResolveHandler` object `handler` is invoked with an iterator object `i` such that the condition `i != iterator_type()` holds. Otherwise it is invoked with `iterator_type()`.

```
iterator_type resolve(implementation_type& impl,
                      const endpoint_type& e, error_code& ec);
```

Effects: Translates an endpoint into a sequence of zero or one `basic_resolver_entry<InternetProtocol>` objects, as if by *POSIX* `getnameinfo()`.

Returns: On success, an iterator object `i` such that the condition `i != iterator_type()` holds. Otherwise `iterator_type()`.

```
template<class ResolveHandler>
  void async_resolve(implementation_type& impl,
                     const endpoint_type& e,
                     ResolveHandler handler);
```

Effects: Initiates an asynchronous operation to translate an endpoint into a sequence of zero or one `basic_resolver_entry<InternetProtocol>` objects, as if by *POSIX* `getnameinfo()`. The operation is performed via the `io_service` object returned by `io_service::service::io_service()` and behaves according to asynchronous operation requirements.

If the operation completes successfully, the `ResolveHandler` object `handler` is invoked with an iterator object `i` such that the condition `i != iterator_type()` holds. Otherwise it is invoked with `iterator_type()`.

5.9.11. Class template `ip::basic_resolver`

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        template<class InternetProtocol, class ResolverService>
        class basic_resolver :
          public basic_io_object<ResolverService>,
          public socket_base
        {
        public:
          // types:
          typedef InternetProtocol protocol_type;
          typedef typename InternetProtocol::endpoint endpoint_type;
          typedef basic_resolver_query<InternetProtocol> query;
          typedef basic_resolver_iterator<InternetProtocol> iterator;

          // constructors:
          explicit basic_resolver(std::tr2::sys::io_service& io_service);

          // members:
          void cancel();
          error_code cancel(error_code& ec);

          iterator resolve(const query& q);
          iterator resolve(const query& q, error_code& ec);

          template <class ResolveHandler>
            void async_resolve(const query& q, ResolveHandler handler);

          iterator resolve(const endpoint_type& e);
          iterator resolve(const endpoint_type& e, error_code& ec);

          template <class ResolveHandler>
            void async_resolve(const endpoint_type& e,
                               ResolveHandler handler);
        };

      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

5.9.11.1. `ip::basic_resolver` constructors

```
explicit basic_resolver(std::tr2::sys::io_service& io_service);
```

Effects: Constructs an object of class `basic_resolver<InternetProtocol, ResolverService>`, initialising the base class with `basic_io_object(io_service)`.

5.9.11.2. `ip::basic_resolver` members

```
void cancel();
error_code cancel(error_code& ec);
```

Returns: `this->service.cancel(this->implementation, ec)`.

```
iterator resolve(const query& q);
iterator resolve(const query& q, error_code& ec);
```

Returns: `this->service.resolve(this->implementation, q, ec)`.

```
template <class ResolveHandler>
  void async_resolve(const query& q, ResolveHandler handler);
```

Effects: Calls `this->service.async_resolve(this->implementation, q, handler)`.

```
iterator resolve(const endpoint_type& e);
iterator resolve(const endpoint_type& e, error_code& ec);
```

Returns: `this->service.resolve(this->implementation, e, ec)`.

```
template <class ResolveHandler>
  void async_resolve(const endpoint_type& e,
                     ResolveHandler handler);
```

Effects: Calls `this->service.async_resolve(this->implementation, e, handler)`.

5.9.12. Host name functions

```
string host_name();
string host_name(error_code&);
```

Returns: The standard host name for the current machine, determined as if by *POSIX* `gethostname()`.

5.9.13. Class `ip::tcp`

The `tcp` class meets the requirements for an `InternetProtocol`.

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        class tcp
        {
        public:
          // types:
          typedef basic_endpoint<tcp> endpoint;
          typedef basic_resolver<tcp> resolver;
          typedef basic_stream_socket<tcp> socket;
          typedef basic_socket_acceptor<tcp> acceptor;
          typedef basic_socket_iostream<tcp> iostream;
          class no_delay;

          // static members:
          static tcp v4();
          static tcp v6();

        private:
          tcp(); // not defined
        };

        // tcp comparisons:
        bool operator==(const tcp& a, const tcp& b);
        bool operator!=(const tcp& a, const tcp& b);

      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

Extensible implementations shall provide the following member functions:

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        class tcp
        {
        public:
          int family() const;
          int type() const;
          int protocol() const;
          // remainder unchanged
        };

      } // namespace ip
    } // namespace sys
```

```
  } // namespace tr2
} // namespace std
```

In the table below, `u` denotes an identifier.

Table--Behaviour of extensible implementations

expression	value
`tcp u(tcp::v4()); u.family();`	`AF_INET`
`tcp u(tcp::v4()); u.type();`	`SOCK_STREAM`
`tcp u(tcp::v4()); u.protocol();`	`IPPROTO_TCP`
`tcp u(tcp::v6()); u.family();`	`AF_INET6`
`tcp u(tcp::v6()); u.type();`	`SOCK_STREAM`
`tcp u(tcp::v6()); u.protocol();`	`IPPROTO_TCP`

[*Note:* The constants `AF_INET`, `AF_INET6` and `SOCK_STREAM` are defined in the *POSIX* header file `sys/socket.h`. The constant `IPPROTO_TCP` is defined in the *POSIX* header file `netinet/in.h`. *—end note*]

5.9.13.1. `ip::tcp` comparisons

`bool operator==(const tcp& a, const tcp& b);`

Returns: A boolean indicating whether two objects of class `tcp` are equal, such that the expression `tcp::v4() == tcp::v4()` is true, the expression `tcp::v6() == tcp::v6()` is true, and the expression `tcp::v4() == tcp::v6()` is false.

`bool operator!=(const tcp& a, const tcp& b);`

Returns: `!(a == b)`.

5.9.14. Class `ip::tcp::no_delay`

The `no_delay` class represents a socket option for disabling the Nagle algorithm for coalescing small segments. It shall be defined as a boolean socket option with the name and values in the table below:

Table--tcp::no_delay boolean socket option

C	*L*	*N*
`tcp::no_delay`	`IPPROTO_TCP`	`TCP_NODELAY`

[*Note:* The constant `IPPROTO_TCP` is defined in the *POSIX* header file `netinet/in.h`. The constant `TCP_NODELAY` is defined in the *POSIX* header file `netinet/tcp.h`. *—end note*]

5.9.15. Class `ip::udp`

The `udp` class meets the requirements for an `InternetProtocol`.

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {

        class udp
        {
        public:
          // types:
          typedef basic_endpoint<udp> endpoint;
          typedef basic_resolver<udp> resolver;
          typedef basic_datagram_socket<udp> socket;

          // static members:
          static udp v4();
          static udp v6();

        private:
          udp(); // not defined
        };

        // udp comparisons:
        bool operator==(const udp& a, const udp& b);
```

```
      bool operator!=(const udp& a, const udp& b);

    } // namespace ip
  } // namespace sys
 } // namespace tr2
} // namespace std
```

Extensible implementations shall provide the following member functions:

```
namespace std {
 namespace tr2 {
  namespace sys {
   namespace ip {

    class udp
    {
    public:
     int family() const;
     int type() const;
     int protocol() const;
     // remainder unchanged
    };

   } // namespace ip
  } // namespace sys
 } // namespace tr2
} // namespace std
```

In the table below, u denotes an identifier.

Table--Behaviour of extensible implementations

expression	value
udp u(udp::v4()); u.family();	AF_INET
udp u(udp::v4()); u.type();	SOCK_DGRAM
udp u(udp::v4()); u.protocol();	IPPROTO_UDP
udp u(udp::v6()); u.family();	AF_INET6
udp u(udp::v6()); u.type();	SOCK_DGRAM
udp u(udp::v6()); u.protocol();	IPPROTO_UDP

[*Note:* The constants AF_INET, AF_INET6 and SOCK_DGRAM are defined in the *POSIX* header file sys/socket.h. The constant IPPROTO_UDP is defined in the *POSIX* header file netinet/in.h. *—end note*]

5.9.15.1. ip::udp comparisons

`bool operator==(const udp& a, const udp& b);`

Returns: A boolean indicating whether two objects of class udp are equal, such that the expression udp::v4() == udp::v4() is true, the expression udp::v6() == udp::v6() is true, and the expression udp::v4() == udp::v6() is false.

`bool operator!=(const udp& a, const udp& b);`

Returns: !(a == b).

5.9.16. Class ip::v6_only

The v6_only class represents a socket option for determining whether a socket created for an IPv6 protocol is restricted to IPv6 communications only. It shall be defined as a boolean socket option with the name and values in the table below:

Table--v6_only boolean socket option

C	*L*	*N*
v6_only	IPPROTO_IPV6	IPV6_V6ONLY

[*Note:* The constants IPPROTO_IPV6 and IPV6_V6ONLY are defined in the *POSIX* header file netinet/in.h. *—end note*]

5.9.17. Class ip::unicast::hops

The hops class represents a socket option for specifying the default number of hops (also known as time-to-live or TTL) on outbound datagrams. It shall be defined as an integral socket option with the name and values in the table below:

Table--hops integral socket option

C	***L***	***N***
`ip::unicast::hops`	`IPPROTO_IPV6` if `p.family() == AF_INET6`, otherwise `IPPROTO_IP`.	`IPV6_UNICAST_HOPS` if `p.family() == AF_INET6`, otherwise `IP_TTL`.

[*Note:* The constants `IPPROTO_IP`, `IPPROTO_IPV6` and `IPV6_UNICAST_HOPS` are defined in the *POSIX* header file `netinet/in.h`. *—end note*]

Where is `IP_TTL` in POSIX?

Constructors for the `hops` class shall throw `out_of_range` if the argument is not in the range `[0, 255]`.

5.9.18. Multicast group management socket options

The `ip::multicast::join_group` and `ip::multicast::leave_group` classes are socket options for multicast group management.

Multicast group management socket option classes satisfy the requirements for `CopyConstructible`, `Assignable`, and `SettableSocketOption`.

[*Example:* Creating a UDP socket and joining a multicast group:

```
// Open an IPv4 UDP socket bound to a specific port.
ip::udp::endpoint ep(ip::udp::v4(), 12345);
ip::udp::socket sock(io_svc, ep);

// Join a multicast group.
ip::address addr = ip::address::from_string("239.255.0.1");
sock.set_option(ip::multicast::join_group(addr));
```

—end example]

Multicast group management socket option classes shall be defined as follows:

```
class C
{
public:
  // constructors:
  C();
  explicit C(const address& multicast_group);
  explicit C(const address_v4& multicast_group,
             const address_v4& network_interface = address_v4::any());
  explicit C(const address_v6& multicast_group,
             unsigned int network_interface = 0);
};
```

Extensible implementations shall provide the following member functions:

```
class C
{
public:
  template<class Protocol> int level(const Protocol& p) const;
  template<class Protocol> int name(const Protocol& p) const;
  template<class Protocol> const unspecified* data(const Protocol& p) const;
  template<class Protocol> size_t size(const Protocol& p) const;
  // remainder unchanged
private:
//ip_mreq v4_value_;    exposition only
//ipv6_mreq v6_value_;  exposition only
};
```

The names and values used in the definition of the multicast group management socket option classes are described in the table below.

Table--Multicast group management socket options

C	***L***	***N***	**description**
`ip::multicast::join_group`	`IPPROTO_IPV6` if `p.family() == AF_INET6`, otherwise `IPPROTO_IP`.	`IPV6_JOIN_GROUP` if `p.family() == AF_INET6`, otherwise `IP_ADD_MEMBERSHIP`.	Used to join a multicast group.

C	***L***	***N***	**description**
`ip::multicast::leave_group`	`IPPROTO_IPV6` if `p.family() == AF_INET6`, otherwise `IPPROTO_IP`.	`IPV6_LEAVE_GROUP` if `p.family() == AF_INET6`, otherwise `IP_DROP_MEMBERSHIP`.	Used to leave a multicast group.

[*Note:* The constants `IPPROTO_IP` and `IPPROTO_IPV6` are defined in the *POSIX* header file `netinet/in.h`. The constants `IPV6_JOIN_GROUP` and `IPV6_LEAVE_GROUP` are defined in the *POSIX* header file `netinet/in.h`. *—end note*]

Where are `IP_ADD_MEMBERSHIP` and `IP_DROP_MEMBERSHIP` in POSIX?

5.9.18.1. Multicast group management socket option constructors

```
C();
```

Effects: For extensible implementations, both `v4_value_` and `v6_value_` are zero-initialised.

```
explicit C(const address& multicast_group);
```

Effects: For extensible implementations, if `multicast_group.is_v6()` is true then `v6_value_.ipv6mr_multiaddr` is initialised to correspond to the IPv6 address returned by `multicast_group.to_v6()`, `v6_value_.ipv6mr_interface` is set to `0`, and `v4_value_` is zero-initialised; otherwise, `v4_value_.imr_multiaddr` is initialised to correspond to the IPv4 address returned by `multicast_group.to_v4()`, `v4_value_.imr_interface` is zero-initialised, and `v6_value_` is zero-initialised.

```
explicit C(const address_v4& multicast_group,
           const address_v4& network_interface = address_v4::any());
```

Effects: For extensible implementations, `v4_value_.imr_multiaddr` is initialised to correspond to the address `multicast_group`, `v4_value_.imr_interface` is initialised to correspond to address `network_interface`, and `v6_value_` is zero-initialised.

```
explicit C(const address_v6& multicast_group,
           unsigned int network_interface = 0);
```

Effects: For extensible implementations, `v6_value_.ipv6mr_multiaddr` is initialised to correspond to the address `multicast_group`, `v6_value_.ipv6mr_interface` is initialised to `network_interface`, and `v4_value_` is zero-initialised.

5.9.18.2. Multicast group management socket option members (extensible implementations)

```
template<class Protocol> int level(const Protocol& p) const;
```

Returns: ***L***.

```
template<class Protocol> int name(const Protocol& p) const;
```

Returns: ***N***.

```
template<class Protocol> const unspecified* data(const Protocol& p) const;
```

Returns: `&v6_value_` if `p.family() == AF_INET6`, otherwise `&v4_value_`.

```
template<class Protocol> size_t size(const Protocol& p) const;
```

Returns: `sizeof(v6_value_)` if `p.family() == AF_INET6`, otherwise `sizeof(v4_value_)`.

5.9.19. Class `ip::multicast::outbound_interface`

The `outbound_interface` class represents a socket option that specifies the network interface to use for outgoing multicast datagrams.

`outbound_interface` satisfies the requirements for `CopyConstructible`, `Assignable`, and `SettableSocketOption`.

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {
        namespace multicast {

          class outbound_interface
```

```
          {
          public:
            // constructors:
            outbound_interface();
            explicit outbound_interface(const address_v4& network_interface);
            explicit outbound_interface(unsigned int network_interface);
          };

        } // namespace multicast
      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

Extensible implementations shall provide the following member functions:

```
namespace std {
  namespace tr2 {
    namespace sys {
      namespace ip {
        namespace multicast {

          class outbound_interface
          {
          public:
            template<class Protocol> int level(const Protocol& p) const;
            template<class Protocol> int name(const Protocol& p) const;
            template<class Protocol> const unspecified* data(const Protocol& p) const;
            template<class Protocol> size_t size(const Protocol& p) const;
            // remainder unchanged
          private:
//          in_addr v4_value_;       exposition only
//          unsigned int v6_value_;  exposition only
          };

        } // namespace multicast
      } // namespace ip
    } // namespace sys
  } // namespace tr2
} // namespace std
```

The `outbound_interface` class is a `SettableSocketOption` only, unlike its POSIX equivalents which are both gettable and settable. This is to avoid the need for additional classes that provide a protocol independent representation of a network interface.

5.9.19.1. `ip::multicast::outbound_interface` constructors

```
outbound_interface();
```

Effects: For extensible implementations, both `v4_value_` and `v6_value_` are zero-initialised.

```
explicit outbound_interface(const address_v4& network_interface);
```

Effects: For extensible implementations, `v4_value_` is initialised to correspond to the IPv4 address `network_interface`, and `v6_value_` is zero-initialised.

```
explicit outbound_interface(unsigned int network_interface);
```

Effects: For extensible implementations, `v6_value_` is initialised to `network_interface`, and `v4_value_` is zero-initialised.

5.9.19.2. `ip::multicast::outbound_interface` members (extensible implementations)

```
template<class Protocol> int level(const Protocol& p) const;
```

Returns: `IPPROTO_IPV6` if `p.family() == AF_INET6`, otherwise `IPPROTO_IP`.

[*Note:* The constants `IPPROTO_IP` and `IPPROTO_IPV6` are defined in the *POSIX* header file `netinet/in.h`. — *end note*]

```
template<class Protocol> int name(const Protocol& p) const;
```

Returns: `IPV6_MULTICAST_IF` if `p.family() == AF_INET6`, otherwise `IP_MULTICAST_IF`.

[*Note:* The constant IPV6_MULTICAST_IF is defined in the *POSIX* header file netinet/in.h. —*end note*]

Where is IP_MULTICAST_IF in POSIX?

```
template<class Protocol> const unspecified* data(const Protocol& p) const;
```

Returns: &v6_value_ if p.family() == AF_INET6, otherwise &v4_value_.

```
template<class Protocol> size_t size(const Protocol& p) const;
```

Returns: sizeof(v6_value_) if p.family() == AF_INET6, otherwise sizeof(v4_value_).

5.9.20. Class ip::multicast::hops

The hops class represents a socket option for specifying the default number of hops (also known as time-to-live or TTL) on outbound multicast datagrams. It shall be defined as an integral socket option with the name and values in the table below:

Table--hops integral socket option

C	*L*	*N*
ip::multicast::hops	IPPROTO_IPV6 if p.family() == AF_INET6, otherwise IPPROTO_IP.	IPV6_MULTICAST_HOPS if p.family() == AF_INET6, otherwise IP_MULTICAST_TTL.

[*Note:* The constants IPPROTO_IP, IPPROTO_IPV6 and IPV6_MULTICAST_HOPS are defined in the *POSIX* header file netinet/in.h. —*end note*]

Where is IP_MULTICAST_TTL in POSIX?

Constructors for the hops class shall throw out_of_range if the argument is not in the range [0, 255].

5.9.21. Class ip::multicast::enable_loopback

The enable_loopback class represents a socket option for determining whether multicast datagrams are delivered back to the local application. It shall be defined as a boolean socket option with the name and values in the table below:

Table--enable_loopback boolean socket option

C	*L*	*N*
ip::multicast::enable_loopback	IPPROTO_IPV6 if p.family() == AF_INET6, otherwise IPPROTO_IP.	IPV6_MULTICAST_LOOP if p.family() == AF_INET6, otherwise IP_MULTICAST_LOOP.

[*Note:* The constants IPPROTO_IP, IPPROTO_IPV6 and IPV6_MULTICAST_LOOP are defined in the *POSIX* header file netinet/in.h. —*end note*]

Where is IP_MULTICAST_LOOP in POSIX?

6. Open Issues

- Impact of trapping signed char implementations on streambuf and iostream support.

7. Acknowledgements

The original ideas behind Boost.Asio were based on the work of Alexander Libman, and were fleshed out in discussions with him over numerous lunches.

The first user of the library was Matthew Nourse, and over the past several years he has provided much valuable feedback on both the library and this proposal, and has been a not entirely unwilling accomplice in resolving many design issues.

Beman Dawes encouraged and guided the development of this proposal. Review feedback was also gratefully received from John Ky and Andrew Kohlhoff.

The library has received feedback in the form of comments, criticisms, contributions, bug reports, suggestions and questions from more than 100 people. Many thanks to all concerned — it has all helped immensely in bringing the interface to maturity. Significant contributions to the design, rationale and implementation were made by Eugene Alterman, Stefan Arentz, Christopher Baus, Dick Bridges, Vygintas Daugmaudis, Beman Dawes, Felipe Magno de Almeida, Giovanni Deretta, Peter Dimov, Caleb Epstein, Jeff Garland, Ion Gatzañaga, Darryl Green, Dirk Griffioen, Indrek Juhani, Jarl Lindrud, Jeremy Maitin-Shepard, Simon Meiklejohn, Cory Nelson, Arvid Norberg, Thorsten Ottosen, Rene Rivera, Roland Schwarz, Andrew

Schweitzer, Arkadiy Vertleyb, Matthew Vogt, Pavel Vozenilek, and Theo Zourzouvillys. Not all of these people agreed with the design choices that have ultimately been selected, but the discussion they provoked was just as important in clarifying and solidifying the thinking behind the design.

Finally, I would like to thank my wife Pauline, whose talents as a software engineer were put to good use in talking through design problems. She also took the time to meticulously read and review this proposal (except this bit).

8. References

[POSIX] *ISO/IEC 9945:2003*, *IEEE Std 1003.1-2001*, and *The Open Group Base Specifications, Issue 6*. Also known as *The Single Unix Specification, Version 3*.

[N1900] Garland, Jeff, *Proposal to Add Date-Time to the C++ Standard Library*, 2006. Updated by N2058.

[N1975] Dawes, Beman, *Filesystem Library Proposal for TR2 (Revision 3)*, 2006

[N2066] Dawes, Beman, *TR2 Diagnostics Enhancements*, 2006

[ACE] Schmidt, Douglas C., *ADAPTIVE Communication Environment*, http://www.cs.wustl.edu/~schmidt/ACE.html

[SYMBIAN] Symbian Ltd, *Sockets Client*, http://www.symbian.com/developer/techlib/v70sdocs/doc_source/reference/cpp/SocketsClient/index.html

[MS-NET] Microsoft Corporation, *.NET Framework Class Library, Socket Class*, http://msdn2.microsoft.com/en-us/library/system.net.sockets.socket.aspx

[ES-API] The Interconnect Software Consortium / The Open Group, *Extended Sockets API (ES-API), Issue 1.0*, 2005, http://opengroup.org/icsc/uploads/40/6415/ES_API_1_0.pdf

[UNPV1] Stevens, W. Richard, *UNIX Network Programming, Volume 1, 2nd Edition*, Prentice Hall, 1998

[POSA2] Schmidt, Douglas C. et al, *Pattern Oriented Software Architecture, Volume 2*, Wiley, 2000

[RFC821] Postel, J., *RFC 821: Simple Mail Transfer Protocol*, 1982, http://www.ietf.org/rfc/rfc0821.txt

[RFC959] Postel, J. and Reynolds, J., *RFC 959: File Transfer Protocol (FTP)*, 1985, http://www.ietf.org/rfc/rfc0959.txt

[RFC2616] Fielding, R. et al, *RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1*, 1999, http://www.ietf.org/rfc/rfc2616.txt

[RFC2732] Hinden, R., Carpenter, B. and Masinter, L., *RFC 2732: Format for Literal IPv6 Addresses in URL's*, 1999, http://www.ietf.org/rfc/rfc2732.txt

[RFC3513] Hinden, R. and Deering, S., *RFC 3513: Internet Protocol Version 6 (IPv6) Addressing Architecture*, 2003, http://www.ietf.org/rfc/rfc3513.txt

---