

asio C++ library

1.10.6

Reference Manual

Copyright © 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015 Christopher M. Kohlhoff

Contents

1 Overview	79
1.1 Rationale	80
1.2 Core Concepts and Functionality	80
1.2.1 Basic Asio Anatomy	81
1.2.2 The Proactor Design Pattern: Concurrency Without Threads	83
1.2.3 Threads and Asio	86
1.2.4 Strands: Use Threads Without Explicit Locking	87
1.2.5 Buffers	87
1.2.6 Streams, Short Reads and Short Writes	90
1.2.7 Reactor-Style Operations	90
1.2.8 Line-Based Operations	91
1.2.9 Custom Memory Allocation	92
1.2.10 Handler Tracking	93
1.2.11 Stackless Coroutines	94
1.2.12 Stackful Coroutines	95
1.3 Networking	96
1.3.1 TCP, UDP and ICMP	96
1.3.2 Support for Other Protocols	98
1.3.3 Socket Iostreams	99
1.3.4 The BSD Socket API and Asio	100
1.4 Timers	102
1.5 Serial Ports	103
1.6 Signal Handling	104
1.7 POSIX-Specific Functionality	104
1.7.1 UNIX Domain Sockets	104
1.7.2 Stream-Oriented File Descriptors	105
1.7.3 Fork	106
1.8 Windows-Specific Functionality	106
1.8.1 Stream-Oriented HANDLEs	106
1.8.2 Random-Access HANDLEs	107
1.8.3 Object HANDLEs	107
1.9 SSL	108
1.10 C++ 2011 Support	109
1.10.1 System Errors and Error Codes	110
1.10.2 Movable I/O Objects	110
1.10.3 Movable Handlers	111
1.10.4 Variadic Templates	111

1.10.5	Array Container	111
1.10.6	Atomics	112
1.10.7	Shared Pointers	112
1.10.8	Chrono	112
1.10.9	Futures	112
1.11	Platform-Specific Implementation Notes	113
2	Using Asio	118
3	Tutorial	123
3.1	Timer.1 - Using a timer synchronously	123
3.1.1	Source listing for Timer.1	124
3.2	Timer.2 - Using a timer asynchronously	125
3.2.1	Source listing for Timer.2	125
3.3	Timer.3 - Binding arguments to a handler	126
3.3.1	Source listing for Timer.3	127
3.4	Timer.4 - Using a member function as a handler	128
3.4.1	Source listing for Timer.4	130
3.5	Timer.5 - Synchronising handlers in multithreaded programs	131
3.5.1	Source listing for Timer.5	132
3.6	Daytime.1 - A synchronous TCP daytime client	134
3.6.1	Source listing for Daytime.1	135
3.7	Daytime.2 - A synchronous TCP daytime server	136
3.7.1	Source listing for Daytime.2	137
3.8	Daytime.3 - An asynchronous TCP daytime server	138
3.8.1	Source listing for Daytime.3	141
3.9	Daytime.4 - A synchronous UDP daytime client	143
3.9.1	Source listing for Daytime.4	144
3.10	Daytime.5 - A synchronous UDP daytime server	145
3.10.1	Source listing for Daytime.5	146
3.11	Daytime.6 - An asynchronous UDP daytime server	147
3.11.1	Source listing for Daytime.6	149
3.12	Daytime.7 - A combined TCP/UDP asynchronous server	150
3.12.1	Source listing for Daytime.7	153
3.13	boost::bind	155
4	Examples	156
4.1	C++03 Examples	156
4.2	C++11 Examples	162

5 Reference	164
5.1 Requirements on asynchronous operations	166
5.2 Accept handler requirements	169
5.3 Buffer-oriented asynchronous random-access read device requirements	169
5.4 Buffer-oriented asynchronous random-access write device requirements	170
5.5 Buffer-oriented asynchronous read stream requirements	171
5.6 Buffer-oriented asynchronous write stream requirements	172
5.7 Buffered handshake handler requirements	173
5.8 Completion handler requirements	174
5.9 Composed connect handler requirements	175
5.10 Connect handler requirements	176
5.11 Constant buffer sequence requirements	176
5.12 Convertible to const buffer requirements	178
5.13 Convertible to mutable buffer requirements	179
5.14 Datagram socket service requirements	180
5.15 Descriptor service requirements	186
5.16 Endpoint requirements	188
5.17 Gettable serial port option requirements	189
5.18 Gettable socket option requirements	190
5.19 Handlers	190
5.20 Handle service requirements	191
5.21 SSL handshake handler requirements	193
5.22 Internet protocol requirements	194
5.23 I/O control command requirements	195
5.24 I/O object service requirements	195
5.25 Mutable buffer sequence requirements	196
5.26 Object handle service requirements	198
5.27 Protocol requirements	199
5.28 Random access handle service requirements	200
5.29 Raw socket service requirements	203
5.30 Read handler requirements	209
5.31 Resolve handler requirements	210
5.32 Resolver service requirements	211
5.33 Sequenced packet socket service requirements	212
5.34 Serial port service requirements	216
5.35 Service requirements	221
5.36 Settable serial port option requirements	222
5.37 Settable socket option requirements	222
5.38 SSL shutdown handler requirements	223

5.39	Signal handler requirements	224
5.40	Signal set service requirements	224
5.41	Socket acceptor service requirements	225
5.42	Socket service requirements	228
5.43	Stream descriptor service requirements	231
5.44	Stream handle service requirements	235
5.45	Stream socket service requirements	239
5.46	Buffer-oriented synchronous random-access read device requirements	243
5.47	Buffer-oriented synchronous random-access write device requirements	244
5.48	Buffer-oriented synchronous read stream requirements	245
5.49	Buffer-oriented synchronous write stream requirements	246
5.50	Time traits requirements	247
5.51	Timer service requirements	248
5.52	Waitable timer service requirements	249
5.53	Wait handler requirements	250
5.54	Wait traits requirements	251
5.55	Write handler requirements	251
5.56	add_service	252
5.57	asio_handler_allocate	253
5.58	asio_handler_deallocate	254
5.59	asio_handler_invoke	254
5.59.1	asio_handler_invoke (1 of 2 overloads)	255
5.59.2	asio_handler_invoke (2 of 2 overloads)	255
5.60	asio_handler_is_continuation	255
5.61	async_connect	256
5.61.1	async_connect (1 of 4 overloads)	257
5.61.2	async_connect (2 of 4 overloads)	258
5.61.3	async_connect (3 of 4 overloads)	259
5.61.4	async_connect (4 of 4 overloads)	261
5.62	async_read	263
5.62.1	async_read (1 of 4 overloads)	264
5.62.2	async_read (2 of 4 overloads)	265
5.62.3	async_read (3 of 4 overloads)	267
5.62.4	async_read (4 of 4 overloads)	268
5.63	async_read_at	269
5.63.1	async_read_at (1 of 4 overloads)	269
5.63.2	async_read_at (2 of 4 overloads)	271
5.63.3	async_read_at (3 of 4 overloads)	272
5.63.4	async_read_at (4 of 4 overloads)	273

5.64	async_read_until	274
5.64.1	async_read_until (1 of 4 overloads)	275
5.64.2	async_read_until (2 of 4 overloads)	277
5.64.3	async_read_until (3 of 4 overloads)	278
5.64.4	async_read_until (4 of 4 overloads)	280
5.65	async_result	282
5.65.1	async_result::async_result	282
5.65.2	async_result::get	283
5.65.3	async_result::type	283
5.66	async_write	283
5.66.1	async_write (1 of 4 overloads)	284
5.66.2	async_write (2 of 4 overloads)	285
5.66.3	async_write (3 of 4 overloads)	286
5.66.4	async_write (4 of 4 overloads)	287
5.67	async_write_at	288
5.67.1	async_write_at (1 of 4 overloads)	289
5.67.2	async_write_at (2 of 4 overloads)	290
5.67.3	async_write_at (3 of 4 overloads)	291
5.67.4	async_write_at (4 of 4 overloads)	292
5.68	basic_datagram_socket	293
5.68.1	basic_datagram_socket::assign	297
5.68.1.1	basic_datagram_socket::assign (1 of 2 overloads)	297
5.68.1.2	basic_datagram_socket::assign (2 of 2 overloads)	298
5.68.2	basic_datagram_socket::async_connect	298
5.68.3	basic_datagram_socket::async_receive	299
5.68.3.1	basic_datagram_socket::async_receive (1 of 2 overloads)	299
5.68.3.2	basic_datagram_socket::async_receive (2 of 2 overloads)	300
5.68.4	basic_datagram_socket::async_receive_from	300
5.68.4.1	basic_datagram_socket::async_receive_from (1 of 2 overloads)	301
5.68.4.2	basic_datagram_socket::async_receive_from (2 of 2 overloads)	302
5.68.5	basic_datagram_socket::async_send	302
5.68.5.1	basic_datagram_socket::async_send (1 of 2 overloads)	303
5.68.5.2	basic_datagram_socket::async_send (2 of 2 overloads)	303
5.68.6	basic_datagram_socket::async_send_to	304
5.68.6.1	basic_datagram_socket::async_send_to (1 of 2 overloads)	304
5.68.6.2	basic_datagram_socket::async_send_to (2 of 2 overloads)	305
5.68.7	basic_datagram_socket::at_mark	306
5.68.7.1	basic_datagram_socket::at_mark (1 of 2 overloads)	306
5.68.7.2	basic_datagram_socket::at_mark (2 of 2 overloads)	306

5.68.8	<code>basic_datagram_socket::available</code>	307
5.68.8.1	<code>basic_datagram_socket::available</code> (1 of 2 overloads)	307
5.68.8.2	<code>basic_datagram_socket::available</code> (2 of 2 overloads)	307
5.68.9	<code>basic_datagram_socket::basic_datagram_socket</code>	308
5.68.9.1	<code>basic_datagram_socket::basic_datagram_socket</code> (1 of 6 overloads)	308
5.68.9.2	<code>basic_datagram_socket::basic_datagram_socket</code> (2 of 6 overloads)	309
5.68.9.3	<code>basic_datagram_socket::basic_datagram_socket</code> (3 of 6 overloads)	309
5.68.9.4	<code>basic_datagram_socket::basic_datagram_socket</code> (4 of 6 overloads)	309
5.68.9.5	<code>basic_datagram_socket::basic_datagram_socket</code> (5 of 6 overloads)	310
5.68.9.6	<code>basic_datagram_socket::basic_datagram_socket</code> (6 of 6 overloads)	310
5.68.10	<code>basic_datagram_socket::bind</code>	311
5.68.10.1	<code>basic_datagram_socket::bind</code> (1 of 2 overloads)	311
5.68.10.2	<code>basic_datagram_socket::bind</code> (2 of 2 overloads)	311
5.68.11	<code>basic_datagram_socket::broadcast</code>	312
5.68.12	<code>basic_datagram_socket::bytes_readable</code>	312
5.68.13	<code>basic_datagram_socket::cancel</code>	313
5.68.13.1	<code>basic_datagram_socket::cancel</code> (1 of 2 overloads)	313
5.68.13.2	<code>basic_datagram_socket::cancel</code> (2 of 2 overloads)	314
5.68.14	<code>basic_datagram_socket::close</code>	314
5.68.14.1	<code>basic_datagram_socket::close</code> (1 of 2 overloads)	314
5.68.14.2	<code>basic_datagram_socket::close</code> (2 of 2 overloads)	315
5.68.15	<code>basic_datagram_socket::connect</code>	315
5.68.15.1	<code>basic_datagram_socket::connect</code> (1 of 2 overloads)	316
5.68.15.2	<code>basic_datagram_socket::connect</code> (2 of 2 overloads)	316
5.68.16	<code>basic_datagram_socket::debug</code>	317
5.68.17	<code>basic_datagram_socket::do_not_route</code>	317
5.68.18	<code>basic_datagram_socket::enable_connection_aborted</code>	318
5.68.19	<code>basic_datagram_socket::endpoint_type</code>	319
5.68.20	<code>basic_datagram_socket::get_implementation</code>	319
5.68.20.1	<code>basic_datagram_socket::get_implementation</code> (1 of 2 overloads)	319
5.68.20.2	<code>basic_datagram_socket::get_implementation</code> (2 of 2 overloads)	319
5.68.21	<code>basic_datagram_socket::get_io_service</code>	319
5.68.22	<code>basic_datagram_socket::get_option</code>	320
5.68.22.1	<code>basic_datagram_socket::get_option</code> (1 of 2 overloads)	320
5.68.22.2	<code>basic_datagram_socket::get_option</code> (2 of 2 overloads)	320
5.68.23	<code>basic_datagram_socket::get_service</code>	321
5.68.23.1	<code>basic_datagram_socket::get_service</code> (1 of 2 overloads)	321
5.68.23.2	<code>basic_datagram_socket::get_service</code> (2 of 2 overloads)	321
5.68.24	<code>basic_datagram_socket::implementation</code>	321

5.68.25	<code>basic_datagram_socket::implementation_type</code>	322
5.68.26	<code>basic_datagram_socket::io_control</code>	322
5.68.26.1	<code>basic_datagram_socket::io_control</code> (1 of 2 overloads)	322
5.68.26.2	<code>basic_datagram_socket::io_control</code> (2 of 2 overloads)	323
5.68.27	<code>basic_datagram_socket::is_open</code>	323
5.68.28	<code>basic_datagram_socket::keep_alive</code>	323
5.68.29	<code>basic_datagram_socket::linger</code>	324
5.68.30	<code>basic_datagram_socket::local_endpoint</code>	325
5.68.30.1	<code>basic_datagram_socket::local_endpoint</code> (1 of 2 overloads)	325
5.68.30.2	<code>basic_datagram_socket::local_endpoint</code> (2 of 2 overloads)	325
5.68.31	<code>basic_datagram_socket::lowest_layer</code>	326
5.68.31.1	<code>basic_datagram_socket::lowest_layer</code> (1 of 2 overloads)	326
5.68.31.2	<code>basic_datagram_socket::lowest_layer</code> (2 of 2 overloads)	326
5.68.32	<code>basic_datagram_socket::lowest_layer_type</code>	327
5.68.33	<code>basic_datagram_socket::max_connections</code>	330
5.68.34	<code>basic_datagram_socket::message_do_not_route</code>	330
5.68.35	<code>basic_datagram_socket::message_end_of_record</code>	330
5.68.36	<code>basic_datagram_socket::message_flags</code>	330
5.68.37	<code>basic_datagram_socket::message_out_of_band</code>	331
5.68.38	<code>basic_datagram_socket::message_peek</code>	331
5.68.39	<code>basic_datagram_socket::native</code>	331
5.68.40	<code>basic_datagram_socket::native_handle</code>	331
5.68.41	<code>basic_datagram_socket::native_handle_type</code>	331
5.68.42	<code>basic_datagram_socket::native_non_blocking</code>	332
5.68.42.1	<code>basic_datagram_socket::native_non_blocking</code> (1 of 3 overloads)	332
5.68.42.2	<code>basic_datagram_socket::native_non_blocking</code> (2 of 3 overloads)	333
5.68.42.3	<code>basic_datagram_socket::native_non_blocking</code> (3 of 3 overloads)	335
5.68.43	<code>basic_datagram_socket::native_type</code>	337
5.68.44	<code>basic_datagram_socket::non_blocking</code>	337
5.68.44.1	<code>basic_datagram_socket::non_blocking</code> (1 of 3 overloads)	337
5.68.44.2	<code>basic_datagram_socket::non_blocking</code> (2 of 3 overloads)	337
5.68.44.3	<code>basic_datagram_socket::non_blocking</code> (3 of 3 overloads)	338
5.68.45	<code>basic_datagram_socket::non_blocking_io</code>	338
5.68.46	<code>basic_datagram_socket::open</code>	339
5.68.46.1	<code>basic_datagram_socket::open</code> (1 of 2 overloads)	339
5.68.46.2	<code>basic_datagram_socket::open</code> (2 of 2 overloads)	339
5.68.47	<code>basic_datagram_socket::operator=</code>	340
5.68.47.1	<code>basic_datagram_socket::operator=</code> (1 of 2 overloads)	340
5.68.47.2	<code>basic_datagram_socket::operator=</code> (2 of 2 overloads)	341

5.68.48	<code>basic_datagram_socket::protocol_type</code>	341
5.68.49	<code>basic_datagram_socket::receive</code>	341
5.68.49.1	<code>basic_datagram_socket::receive</code> (1 of 3 overloads)	342
5.68.49.2	<code>basic_datagram_socket::receive</code> (2 of 3 overloads)	342
5.68.49.3	<code>basic_datagram_socket::receive</code> (3 of 3 overloads)	343
5.68.50	<code>basic_datagram_socket::receive_buffer_size</code>	343
5.68.51	<code>basic_datagram_socket::receive_from</code>	344
5.68.51.1	<code>basic_datagram_socket::receive_from</code> (1 of 3 overloads)	344
5.68.51.2	<code>basic_datagram_socket::receive_from</code> (2 of 3 overloads)	345
5.68.51.3	<code>basic_datagram_socket::receive_from</code> (3 of 3 overloads)	346
5.68.52	<code>basic_datagram_socket::receive_low_watermark</code>	346
5.68.53	<code>basic_datagram_socket::remote_endpoint</code>	347
5.68.53.1	<code>basic_datagram_socket::remote_endpoint</code> (1 of 2 overloads)	347
5.68.53.2	<code>basic_datagram_socket::remote_endpoint</code> (2 of 2 overloads)	347
5.68.54	<code>basic_datagram_socket::reuse_address</code>	348
5.68.55	<code>basic_datagram_socket::send</code>	348
5.68.55.1	<code>basic_datagram_socket::send</code> (1 of 3 overloads)	349
5.68.55.2	<code>basic_datagram_socket::send</code> (2 of 3 overloads)	350
5.68.55.3	<code>basic_datagram_socket::send</code> (3 of 3 overloads)	350
5.68.56	<code>basic_datagram_socket::send_buffer_size</code>	351
5.68.57	<code>basic_datagram_socket::send_low_watermark</code>	351
5.68.58	<code>basic_datagram_socket::send_to</code>	352
5.68.58.1	<code>basic_datagram_socket::send_to</code> (1 of 3 overloads)	352
5.68.58.2	<code>basic_datagram_socket::send_to</code> (2 of 3 overloads)	353
5.68.58.3	<code>basic_datagram_socket::send_to</code> (3 of 3 overloads)	354
5.68.59	<code>basic_datagram_socket::service</code>	354
5.68.60	<code>basic_datagram_socket::service_type</code>	354
5.68.61	<code>basic_datagram_socket::set_option</code>	355
5.68.61.1	<code>basic_datagram_socket::set_option</code> (1 of 2 overloads)	355
5.68.61.2	<code>basic_datagram_socket::set_option</code> (2 of 2 overloads)	356
5.68.62	<code>basic_datagram_socket::shutdown</code>	356
5.68.62.1	<code>basic_datagram_socket::shutdown</code> (1 of 2 overloads)	356
5.68.62.2	<code>basic_datagram_socket::shutdown</code> (2 of 2 overloads)	357
5.68.63	<code>basic_datagram_socket::shutdown_type</code>	357
5.69	<code>basic_deadline_timer</code>	358
5.69.1	<code>basic_deadline_timer::async_wait</code>	361
5.69.2	<code>basic_deadline_timer::basic_deadline_timer</code>	361
5.69.2.1	<code>basic_deadline_timer::basic_deadline_timer</code> (1 of 3 overloads)	361
5.69.2.2	<code>basic_deadline_timer::basic_deadline_timer</code> (2 of 3 overloads)	362

5.69.2.3	<code>basic_deadline_timer::basic_deadline_timer</code> (3 of 3 overloads)	362
5.69.3	<code>basic_deadline_timer::cancel</code>	362
5.69.3.1	<code>basic_deadline_timer::cancel</code> (1 of 2 overloads)	362
5.69.3.2	<code>basic_deadline_timer::cancel</code> (2 of 2 overloads)	363
5.69.4	<code>basic_deadline_timer::cancel_one</code>	364
5.69.4.1	<code>basic_deadline_timer::cancel_one</code> (1 of 2 overloads)	364
5.69.4.2	<code>basic_deadline_timer::cancel_one</code> (2 of 2 overloads)	364
5.69.5	<code>basic_deadline_timer::duration_type</code>	365
5.69.6	<code>basic_deadline_timer::expires_at</code>	365
5.69.6.1	<code>basic_deadline_timer::expires_at</code> (1 of 3 overloads)	365
5.69.6.2	<code>basic_deadline_timer::expires_at</code> (2 of 3 overloads)	366
5.69.6.3	<code>basic_deadline_timer::expires_at</code> (3 of 3 overloads)	366
5.69.7	<code>basic_deadline_timer::expires_from_now</code>	367
5.69.7.1	<code>basic_deadline_timer::expires_from_now</code> (1 of 3 overloads)	367
5.69.7.2	<code>basic_deadline_timer::expires_from_now</code> (2 of 3 overloads)	367
5.69.7.3	<code>basic_deadline_timer::expires_from_now</code> (3 of 3 overloads)	368
5.69.8	<code>basic_deadline_timer::get_implementation</code>	368
5.69.8.1	<code>basic_deadline_timer::get_implementation</code> (1 of 2 overloads)	369
5.69.8.2	<code>basic_deadline_timer::get_implementation</code> (2 of 2 overloads)	369
5.69.9	<code>basic_deadline_timer::get_io_service</code>	369
5.69.10	<code>basic_deadline_timer::get_service</code>	369
5.69.10.1	<code>basic_deadline_timer::get_service</code> (1 of 2 overloads)	369
5.69.10.2	<code>basic_deadline_timer::get_service</code> (2 of 2 overloads)	369
5.69.11	<code>basic_deadline_timer::implementation</code>	370
5.69.12	<code>basic_deadline_timer::implementation_type</code>	370
5.69.13	<code>basic_deadline_timer::service</code>	370
5.69.14	<code>basic_deadline_timer::service_type</code>	370
5.69.15	<code>basic_deadline_timer::time_type</code>	370
5.69.16	<code>basic_deadline_timer::traits_type</code>	371
5.69.17	<code>basic_deadline_timer::wait</code>	371
5.69.17.1	<code>basic_deadline_timer::wait</code> (1 of 2 overloads)	371
5.69.17.2	<code>basic_deadline_timer::wait</code> (2 of 2 overloads)	371
5.70	<code>basic_io_object</code>	372
5.70.1	<code>basic_io_object::basic_io_object</code>	373
5.70.1.1	<code>basic_io_object::basic_io_object</code> (1 of 2 overloads)	373
5.70.1.2	<code>basic_io_object::basic_io_object</code> (2 of 2 overloads)	373
5.70.2	<code>basic_io_object::get_implementation</code>	374
5.70.2.1	<code>basic_io_object::get_implementation</code> (1 of 2 overloads)	374
5.70.2.2	<code>basic_io_object::get_implementation</code> (2 of 2 overloads)	374

5.70.3	<code>basic_io_object::get_io_service</code>	374
5.70.4	<code>basic_io_object::get_service</code>	374
5.70.4.1	<code>basic_io_object::get_service</code> (1 of 2 overloads)	374
5.70.4.2	<code>basic_io_object::get_service</code> (2 of 2 overloads)	374
5.70.5	<code>basic_io_object::implementation</code>	375
5.70.6	<code>basic_io_object::implementation_type</code>	375
5.70.7	<code>basic_io_object::operator=</code>	375
5.70.8	<code>basic_io_object::service</code>	375
5.70.9	<code>basic_io_object::service_type</code>	375
5.70.10	<code>basic_io_object::~basic_io_object</code>	376
5.71	<code>basic_raw_socket</code>	376
5.71.1	<code>basic_raw_socket::assign</code>	380
5.71.1.1	<code>basic_raw_socket::assign</code> (1 of 2 overloads)	380
5.71.1.2	<code>basic_raw_socket::assign</code> (2 of 2 overloads)	380
5.71.2	<code>basic_raw_socket::async_connect</code>	380
5.71.3	<code>basic_raw_socket::async_receive</code>	381
5.71.3.1	<code>basic_raw_socket::async_receive</code> (1 of 2 overloads)	382
5.71.3.2	<code>basic_raw_socket::async_receive</code> (2 of 2 overloads)	382
5.71.4	<code>basic_raw_socket::async_receive_from</code>	383
5.71.4.1	<code>basic_raw_socket::async_receive_from</code> (1 of 2 overloads)	383
5.71.4.2	<code>basic_raw_socket::async_receive_from</code> (2 of 2 overloads)	384
5.71.5	<code>basic_raw_socket::async_send</code>	385
5.71.5.1	<code>basic_raw_socket::async_send</code> (1 of 2 overloads)	385
5.71.5.2	<code>basic_raw_socket::async_send</code> (2 of 2 overloads)	386
5.71.6	<code>basic_raw_socket::async_send_to</code>	387
5.71.6.1	<code>basic_raw_socket::async_send_to</code> (1 of 2 overloads)	387
5.71.6.2	<code>basic_raw_socket::async_send_to</code> (2 of 2 overloads)	388
5.71.7	<code>basic_raw_socket::at_mark</code>	388
5.71.7.1	<code>basic_raw_socket::at_mark</code> (1 of 2 overloads)	389
5.71.7.2	<code>basic_raw_socket::at_mark</code> (2 of 2 overloads)	389
5.71.8	<code>basic_raw_socket::available</code>	389
5.71.8.1	<code>basic_raw_socket::available</code> (1 of 2 overloads)	389
5.71.8.2	<code>basic_raw_socket::available</code> (2 of 2 overloads)	390
5.71.9	<code>basic_raw_socket::basic_raw_socket</code>	390
5.71.9.1	<code>basic_raw_socket::basic_raw_socket</code> (1 of 6 overloads)	391
5.71.9.2	<code>basic_raw_socket::basic_raw_socket</code> (2 of 6 overloads)	391
5.71.9.3	<code>basic_raw_socket::basic_raw_socket</code> (3 of 6 overloads)	392
5.71.9.4	<code>basic_raw_socket::basic_raw_socket</code> (4 of 6 overloads)	392
5.71.9.5	<code>basic_raw_socket::basic_raw_socket</code> (5 of 6 overloads)	392

5.71.9.6	<code>basic_raw_socket::basic_raw_socket</code> (6 of 6 overloads)	393
5.71.10	<code>basic_raw_socket::bind</code>	393
5.71.10.1	<code>basic_raw_socket::bind</code> (1 of 2 overloads)	393
5.71.10.2	<code>basic_raw_socket::bind</code> (2 of 2 overloads)	394
5.71.11	<code>basic_raw_socket::broadcast</code>	394
5.71.12	<code>basic_raw_socket::bytes_readable</code>	395
5.71.13	<code>basic_raw_socket::cancel</code>	395
5.71.13.1	<code>basic_raw_socket::cancel</code> (1 of 2 overloads)	396
5.71.13.2	<code>basic_raw_socket::cancel</code> (2 of 2 overloads)	396
5.71.14	<code>basic_raw_socket::close</code>	397
5.71.14.1	<code>basic_raw_socket::close</code> (1 of 2 overloads)	397
5.71.14.2	<code>basic_raw_socket::close</code> (2 of 2 overloads)	398
5.71.15	<code>basic_raw_socket::connect</code>	398
5.71.15.1	<code>basic_raw_socket::connect</code> (1 of 2 overloads)	398
5.71.15.2	<code>basic_raw_socket::connect</code> (2 of 2 overloads)	399
5.71.16	<code>basic_raw_socket::debug</code>	399
5.71.17	<code>basic_raw_socket::do_not_route</code>	400
5.71.18	<code>basic_raw_socket::enable_connection_aborted</code>	401
5.71.19	<code>basic_raw_socket::endpoint_type</code>	401
5.71.20	<code>basic_raw_socket::get_implementation</code>	401
5.71.20.1	<code>basic_raw_socket::get_implementation</code> (1 of 2 overloads)	402
5.71.20.2	<code>basic_raw_socket::get_implementation</code> (2 of 2 overloads)	402
5.71.21	<code>basic_raw_socket::get_io_service</code>	402
5.71.22	<code>basic_raw_socket::get_option</code>	402
5.71.22.1	<code>basic_raw_socket::get_option</code> (1 of 2 overloads)	402
5.71.22.2	<code>basic_raw_socket::get_option</code> (2 of 2 overloads)	403
5.71.23	<code>basic_raw_socket::get_service</code>	404
5.71.23.1	<code>basic_raw_socket::get_service</code> (1 of 2 overloads)	404
5.71.23.2	<code>basic_raw_socket::get_service</code> (2 of 2 overloads)	404
5.71.24	<code>basic_raw_socket::implementation</code>	404
5.71.25	<code>basic_raw_socket::implementation_type</code>	404
5.71.26	<code>basic_raw_socket::io_control</code>	404
5.71.26.1	<code>basic_raw_socket::io_control</code> (1 of 2 overloads)	405
5.71.26.2	<code>basic_raw_socket::io_control</code> (2 of 2 overloads)	405
5.71.27	<code>basic_raw_socket::is_open</code>	406
5.71.28	<code>basic_raw_socket::keep_alive</code>	406
5.71.29	<code>basic_raw_socket::linger</code>	407
5.71.30	<code>basic_raw_socket::local_endpoint</code>	407
5.71.30.1	<code>basic_raw_socket::local_endpoint</code> (1 of 2 overloads)	407

5.71.30.2	<code>basic_raw_socket::local_endpoint</code> (2 of 2 overloads)	408
5.71.31	<code>basic_raw_socket::lowest_layer</code>	408
5.71.31.1	<code>basic_raw_socket::lowest_layer</code> (1 of 2 overloads)	409
5.71.31.2	<code>basic_raw_socket::lowest_layer</code> (2 of 2 overloads)	409
5.71.32	<code>basic_raw_socket::lowest_layer_type</code>	409
5.71.33	<code>basic_raw_socket::max_connections</code>	413
5.71.34	<code>basic_raw_socket::message_do_not_route</code>	413
5.71.35	<code>basic_raw_socket::message_end_of_record</code>	413
5.71.36	<code>basic_raw_socket::message_flags</code>	413
5.71.37	<code>basic_raw_socket::message_out_of_band</code>	413
5.71.38	<code>basic_raw_socket::message_peek</code>	414
5.71.39	<code>basic_raw_socket::native</code>	414
5.71.40	<code>basic_raw_socket::native_handle</code>	414
5.71.41	<code>basic_raw_socket::native_handle_type</code>	414
5.71.42	<code>basic_raw_socket::native_non_blocking</code>	414
5.71.42.1	<code>basic_raw_socket::native_non_blocking</code> (1 of 3 overloads)	415
5.71.42.2	<code>basic_raw_socket::native_non_blocking</code> (2 of 3 overloads)	416
5.71.42.3	<code>basic_raw_socket::native_non_blocking</code> (3 of 3 overloads)	418
5.71.43	<code>basic_raw_socket::native_type</code>	419
5.71.44	<code>basic_raw_socket::non_blocking</code>	419
5.71.44.1	<code>basic_raw_socket::non_blocking</code> (1 of 3 overloads)	420
5.71.44.2	<code>basic_raw_socket::non_blocking</code> (2 of 3 overloads)	420
5.71.44.3	<code>basic_raw_socket::non_blocking</code> (3 of 3 overloads)	421
5.71.45	<code>basic_raw_socket::non_blocking_io</code>	421
5.71.46	<code>basic_raw_socket::open</code>	421
5.71.46.1	<code>basic_raw_socket::open</code> (1 of 2 overloads)	422
5.71.46.2	<code>basic_raw_socket::open</code> (2 of 2 overloads)	422
5.71.47	<code>basic_raw_socket::operator=</code>	423
5.71.47.1	<code>basic_raw_socket::operator=</code> (1 of 2 overloads)	423
5.71.47.2	<code>basic_raw_socket::operator=</code> (2 of 2 overloads)	423
5.71.48	<code>basic_raw_socket::protocol_type</code>	424
5.71.49	<code>basic_raw_socket::receive</code>	424
5.71.49.1	<code>basic_raw_socket::receive</code> (1 of 3 overloads)	424
5.71.49.2	<code>basic_raw_socket::receive</code> (2 of 3 overloads)	425
5.71.49.3	<code>basic_raw_socket::receive</code> (3 of 3 overloads)	426
5.71.50	<code>basic_raw_socket::receive_buffer_size</code>	426
5.71.51	<code>basic_raw_socket::receive_from</code>	427
5.71.51.1	<code>basic_raw_socket::receive_from</code> (1 of 3 overloads)	427
5.71.51.2	<code>basic_raw_socket::receive_from</code> (2 of 3 overloads)	428

5.71.51.3	<code>basic_raw_socket::receive_from</code> (3 of 3 overloads)	428
5.71.52	<code>basic_raw_socket::receive_low_watermark</code>	429
5.71.53	<code>basic_raw_socket::remote_endpoint</code>	429
5.71.53.1	<code>basic_raw_socket::remote_endpoint</code> (1 of 2 overloads)	430
5.71.53.2	<code>basic_raw_socket::remote_endpoint</code> (2 of 2 overloads)	430
5.71.54	<code>basic_raw_socket::reuse_address</code>	431
5.71.55	<code>basic_raw_socket::send</code>	431
5.71.55.1	<code>basic_raw_socket::send</code> (1 of 3 overloads)	432
5.71.55.2	<code>basic_raw_socket::send</code> (2 of 3 overloads)	432
5.71.55.3	<code>basic_raw_socket::send</code> (3 of 3 overloads)	433
5.71.56	<code>basic_raw_socket::send_buffer_size</code>	433
5.71.57	<code>basic_raw_socket::send_low_watermark</code>	434
5.71.58	<code>basic_raw_socket::send_to</code>	435
5.71.58.1	<code>basic_raw_socket::send_to</code> (1 of 3 overloads)	435
5.71.58.2	<code>basic_raw_socket::send_to</code> (2 of 3 overloads)	436
5.71.58.3	<code>basic_raw_socket::send_to</code> (3 of 3 overloads)	436
5.71.59	<code>basic_raw_socket::service</code>	437
5.71.60	<code>basic_raw_socket::service_type</code>	437
5.71.61	<code>basic_raw_socket::set_option</code>	437
5.71.61.1	<code>basic_raw_socket::set_option</code> (1 of 2 overloads)	438
5.71.61.2	<code>basic_raw_socket::set_option</code> (2 of 2 overloads)	438
5.71.62	<code>basic_raw_socket::shutdown</code>	439
5.71.62.1	<code>basic_raw_socket::shutdown</code> (1 of 2 overloads)	439
5.71.62.2	<code>basic_raw_socket::shutdown</code> (2 of 2 overloads)	440
5.71.63	<code>basic_raw_socket::shutdown_type</code>	440
5.72	<code>basic_seq_packet_socket</code>	440
5.72.1	<code>basic_seq_packet_socket::assign</code>	444
5.72.1.1	<code>basic_seq_packet_socket::assign</code> (1 of 2 overloads)	444
5.72.1.2	<code>basic_seq_packet_socket::assign</code> (2 of 2 overloads)	444
5.72.2	<code>basic_seq_packet_socket::async_connect</code>	445
5.72.3	<code>basic_seq_packet_socket::async_receive</code>	445
5.72.3.1	<code>basic_seq_packet_socket::async_receive</code> (1 of 2 overloads)	446
5.72.3.2	<code>basic_seq_packet_socket::async_receive</code> (2 of 2 overloads)	447
5.72.4	<code>basic_seq_packet_socket::async_send</code>	447
5.72.5	<code>basic_seq_packet_socket::at_mark</code>	448
5.72.5.1	<code>basic_seq_packet_socket::at_mark</code> (1 of 2 overloads)	448
5.72.5.2	<code>basic_seq_packet_socket::at_mark</code> (2 of 2 overloads)	449
5.72.6	<code>basic_seq_packet_socket::available</code>	449
5.72.6.1	<code>basic_seq_packet_socket::available</code> (1 of 2 overloads)	449

5.72.6.2	<code>basic_seq_packet_socket::available</code> (2 of 2 overloads)	449
5.72.7	<code>basic_seq_packet_socket::basic_seq_packet_socket</code>	450
5.72.7.1	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (1 of 6 overloads)	451
5.72.7.2	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (2 of 6 overloads)	451
5.72.7.3	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (3 of 6 overloads)	451
5.72.7.4	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (4 of 6 overloads)	452
5.72.7.5	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (5 of 6 overloads)	452
5.72.7.6	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (6 of 6 overloads)	452
5.72.8	<code>basic_seq_packet_socket::bind</code>	453
5.72.8.1	<code>basic_seq_packet_socket::bind</code> (1 of 2 overloads)	453
5.72.8.2	<code>basic_seq_packet_socket::bind</code> (2 of 2 overloads)	454
5.72.9	<code>basic_seq_packet_socket::broadcast</code>	454
5.72.10	<code>basic_seq_packet_socket::bytes_readable</code>	455
5.72.11	<code>basic_seq_packet_socket::cancel</code>	455
5.72.11.1	<code>basic_seq_packet_socket::cancel</code> (1 of 2 overloads)	455
5.72.11.2	<code>basic_seq_packet_socket::cancel</code> (2 of 2 overloads)	456
5.72.12	<code>basic_seq_packet_socket::close</code>	457
5.72.12.1	<code>basic_seq_packet_socket::close</code> (1 of 2 overloads)	457
5.72.12.2	<code>basic_seq_packet_socket::close</code> (2 of 2 overloads)	457
5.72.13	<code>basic_seq_packet_socket::connect</code>	458
5.72.13.1	<code>basic_seq_packet_socket::connect</code> (1 of 2 overloads)	458
5.72.13.2	<code>basic_seq_packet_socket::connect</code> (2 of 2 overloads)	459
5.72.14	<code>basic_seq_packet_socket::debug</code>	459
5.72.15	<code>basic_seq_packet_socket::do_not_route</code>	460
5.72.16	<code>basic_seq_packet_socket::enable_connection_aborted</code>	460
5.72.17	<code>basic_seq_packet_socket::endpoint_type</code>	461
5.72.18	<code>basic_seq_packet_socket::get_implementation</code>	461
5.72.18.1	<code>basic_seq_packet_socket::get_implementation</code> (1 of 2 overloads)	461
5.72.18.2	<code>basic_seq_packet_socket::get_implementation</code> (2 of 2 overloads)	462
5.72.19	<code>basic_seq_packet_socket::get_io_service</code>	462
5.72.20	<code>basic_seq_packet_socket::get_option</code>	462
5.72.20.1	<code>basic_seq_packet_socket::get_option</code> (1 of 2 overloads)	462
5.72.20.2	<code>basic_seq_packet_socket::get_option</code> (2 of 2 overloads)	463
5.72.21	<code>basic_seq_packet_socket::get_service</code>	463
5.72.21.1	<code>basic_seq_packet_socket::get_service</code> (1 of 2 overloads)	464
5.72.21.2	<code>basic_seq_packet_socket::get_service</code> (2 of 2 overloads)	464
5.72.22	<code>basic_seq_packet_socket::implementation</code>	464
5.72.23	<code>basic_seq_packet_socket::implementation_type</code>	464
5.72.24	<code>basic_seq_packet_socket::io_control</code>	464

5.72.24.1	basic_seq_packet_socket::io_control (1 of 2 overloads)	465
5.72.24.2	basic_seq_packet_socket::io_control (2 of 2 overloads)	465
5.72.25	basic_seq_packet_socket::is_open	466
5.72.26	basic_seq_packet_socket::keep_alive	466
5.72.27	basic_seq_packet_socket::linger	467
5.72.28	basic_seq_packet_socket::local_endpoint	467
5.72.28.1	basic_seq_packet_socket::local_endpoint (1 of 2 overloads)	467
5.72.28.2	basic_seq_packet_socket::local_endpoint (2 of 2 overloads)	468
5.72.29	basic_seq_packet_socket::lowest_layer	468
5.72.29.1	basic_seq_packet_socket::lowest_layer (1 of 2 overloads)	469
5.72.29.2	basic_seq_packet_socket::lowest_layer (2 of 2 overloads)	469
5.72.30	basic_seq_packet_socket::lowest_layer_type	469
5.72.31	basic_seq_packet_socket::max_connections	473
5.72.32	basic_seq_packet_socket::message_do_not_route	473
5.72.33	basic_seq_packet_socket::message_end_of_record	473
5.72.34	basic_seq_packet_socket::message_flags	473
5.72.35	basic_seq_packet_socket::message_out_of_band	473
5.72.36	basic_seq_packet_socket::message_peek	474
5.72.37	basic_seq_packet_socket::native	474
5.72.38	basic_seq_packet_socket::native_handle	474
5.72.39	basic_seq_packet_socket::native_handle_type	474
5.72.40	basic_seq_packet_socket::native_non_blocking	474
5.72.40.1	basic_seq_packet_socket::native_non_blocking (1 of 3 overloads)	475
5.72.40.2	basic_seq_packet_socket::native_non_blocking (2 of 3 overloads)	476
5.72.40.3	basic_seq_packet_socket::native_non_blocking (3 of 3 overloads)	478
5.72.41	basic_seq_packet_socket::native_type	479
5.72.42	basic_seq_packet_socket::non_blocking	479
5.72.42.1	basic_seq_packet_socket::non_blocking (1 of 3 overloads)	480
5.72.42.2	basic_seq_packet_socket::non_blocking (2 of 3 overloads)	480
5.72.42.3	basic_seq_packet_socket::non_blocking (3 of 3 overloads)	481
5.72.43	basic_seq_packet_socket::non_blocking_io	481
5.72.44	basic_seq_packet_socket::open	481
5.72.44.1	basic_seq_packet_socket::open (1 of 2 overloads)	482
5.72.44.2	basic_seq_packet_socket::open (2 of 2 overloads)	482
5.72.45	basic_seq_packet_socket::operator=	483
5.72.45.1	basic_seq_packet_socket::operator= (1 of 2 overloads)	483
5.72.45.2	basic_seq_packet_socket::operator= (2 of 2 overloads)	483
5.72.46	basic_seq_packet_socket::protocol_type	484
5.72.47	basic_seq_packet_socket::receive	484

5.72.47.1	<code>basic_seq_packet_socket::receive</code> (1 of 3 overloads)	484
5.72.47.2	<code>basic_seq_packet_socket::receive</code> (2 of 3 overloads)	485
5.72.47.3	<code>basic_seq_packet_socket::receive</code> (3 of 3 overloads)	486
5.72.48	<code>basic_seq_packet_socket::receive_buffer_size</code>	487
5.72.49	<code>basic_seq_packet_socket::receive_low_watermark</code>	487
5.72.50	<code>basic_seq_packet_socket::remote_endpoint</code>	488
5.72.50.1	<code>basic_seq_packet_socket::remote_endpoint</code> (1 of 2 overloads)	488
5.72.50.2	<code>basic_seq_packet_socket::remote_endpoint</code> (2 of 2 overloads)	488
5.72.51	<code>basic_seq_packet_socket::reuse_address</code>	489
5.72.52	<code>basic_seq_packet_socket::send</code>	489
5.72.52.1	<code>basic_seq_packet_socket::send</code> (1 of 2 overloads)	490
5.72.52.2	<code>basic_seq_packet_socket::send</code> (2 of 2 overloads)	491
5.72.53	<code>basic_seq_packet_socket::send_buffer_size</code>	491
5.72.54	<code>basic_seq_packet_socket::send_low_watermark</code>	492
5.72.55	<code>basic_seq_packet_socket::service</code>	492
5.72.56	<code>basic_seq_packet_socket::service_type</code>	492
5.72.57	<code>basic_seq_packet_socket::set_option</code>	493
5.72.57.1	<code>basic_seq_packet_socket::set_option</code> (1 of 2 overloads)	493
5.72.57.2	<code>basic_seq_packet_socket::set_option</code> (2 of 2 overloads)	494
5.72.58	<code>basic_seq_packet_socket::shutdown</code>	494
5.72.58.1	<code>basic_seq_packet_socket::shutdown</code> (1 of 2 overloads)	494
5.72.58.2	<code>basic_seq_packet_socket::shutdown</code> (2 of 2 overloads)	495
5.72.59	<code>basic_seq_packet_socket::shutdown_type</code>	495
5.73	<code>basic_serial_port</code>	496
5.73.1	<code>basic_serial_port::assign</code>	498
5.73.1.1	<code>basic_serial_port::assign</code> (1 of 2 overloads)	498
5.73.1.2	<code>basic_serial_port::assign</code> (2 of 2 overloads)	498
5.73.2	<code>basic_serial_port::async_read_some</code>	498
5.73.3	<code>basic_serial_port::async_write_some</code>	499
5.73.4	<code>basic_serial_port::basic_serial_port</code>	500
5.73.4.1	<code>basic_serial_port::basic_serial_port</code> (1 of 5 overloads)	500
5.73.4.2	<code>basic_serial_port::basic_serial_port</code> (2 of 5 overloads)	501
5.73.4.3	<code>basic_serial_port::basic_serial_port</code> (3 of 5 overloads)	501
5.73.4.4	<code>basic_serial_port::basic_serial_port</code> (4 of 5 overloads)	501
5.73.4.5	<code>basic_serial_port::basic_serial_port</code> (5 of 5 overloads)	502
5.73.5	<code>basic_serial_port::cancel</code>	502
5.73.5.1	<code>basic_serial_port::cancel</code> (1 of 2 overloads)	502
5.73.5.2	<code>basic_serial_port::cancel</code> (2 of 2 overloads)	502
5.73.6	<code>basic_serial_port::close</code>	503

5.73.6.1	<code>basic_serial_port::close</code> (1 of 2 overloads)	503
5.73.6.2	<code>basic_serial_port::close</code> (2 of 2 overloads)	503
5.73.7	<code>basic_serial_port::get_implementation</code>	503
5.73.7.1	<code>basic_serial_port::get_implementation</code> (1 of 2 overloads)	504
5.73.7.2	<code>basic_serial_port::get_implementation</code> (2 of 2 overloads)	504
5.73.8	<code>basic_serial_port::get_io_service</code>	504
5.73.9	<code>basic_serial_port::get_option</code>	504
5.73.9.1	<code>basic_serial_port::get_option</code> (1 of 2 overloads)	504
5.73.9.2	<code>basic_serial_port::get_option</code> (2 of 2 overloads)	505
5.73.10	<code>basic_serial_port::get_service</code>	505
5.73.10.1	<code>basic_serial_port::get_service</code> (1 of 2 overloads)	505
5.73.10.2	<code>basic_serial_port::get_service</code> (2 of 2 overloads)	505
5.73.11	<code>basic_serial_port::implementation</code>	506
5.73.12	<code>basic_serial_port::implementation_type</code>	506
5.73.13	<code>basic_serial_port::is_open</code>	506
5.73.14	<code>basic_serial_port::lowest_layer</code>	506
5.73.14.1	<code>basic_serial_port::lowest_layer</code> (1 of 2 overloads)	506
5.73.14.2	<code>basic_serial_port::lowest_layer</code> (2 of 2 overloads)	507
5.73.15	<code>basic_serial_port::lowest_layer_type</code>	507
5.73.16	<code>basic_serial_port::native</code>	509
5.73.17	<code>basic_serial_port::native_handle</code>	509
5.73.18	<code>basic_serial_port::native_handle_type</code>	509
5.73.19	<code>basic_serial_port::native_type</code>	509
5.73.20	<code>basic_serial_port::open</code>	510
5.73.20.1	<code>basic_serial_port::open</code> (1 of 2 overloads)	510
5.73.20.2	<code>basic_serial_port::open</code> (2 of 2 overloads)	510
5.73.21	<code>basic_serial_port::operator=</code>	510
5.73.22	<code>basic_serial_port::read_some</code>	511
5.73.22.1	<code>basic_serial_port::read_some</code> (1 of 2 overloads)	511
5.73.22.2	<code>basic_serial_port::read_some</code> (2 of 2 overloads)	512
5.73.23	<code>basic_serial_port::send_break</code>	512
5.73.23.1	<code>basic_serial_port::send_break</code> (1 of 2 overloads)	513
5.73.23.2	<code>basic_serial_port::send_break</code> (2 of 2 overloads)	513
5.73.24	<code>basic_serial_port::service</code>	513
5.73.25	<code>basic_serial_port::service_type</code>	513
5.73.26	<code>basic_serial_port::set_option</code>	514
5.73.26.1	<code>basic_serial_port::set_option</code> (1 of 2 overloads)	514
5.73.26.2	<code>basic_serial_port::set_option</code> (2 of 2 overloads)	514
5.73.27	<code>basic_serial_port::write_some</code>	515

5.73.27.1	<code>basic_serial_port::write_some</code> (1 of 2 overloads)	515
5.73.27.2	<code>basic_serial_port::write_some</code> (2 of 2 overloads)	516
5.74	<code>basic_signal_set</code>	516
5.74.1	<code>basic_signal_set::add</code>	518
5.74.1.1	<code>basic_signal_set::add</code> (1 of 2 overloads)	519
5.74.1.2	<code>basic_signal_set::add</code> (2 of 2 overloads)	519
5.74.2	<code>basic_signal_set::async_wait</code>	519
5.74.3	<code>basic_signal_set::basic_signal_set</code>	520
5.74.3.1	<code>basic_signal_set::basic_signal_set</code> (1 of 4 overloads)	520
5.74.3.2	<code>basic_signal_set::basic_signal_set</code> (2 of 4 overloads)	521
5.74.3.3	<code>basic_signal_set::basic_signal_set</code> (3 of 4 overloads)	521
5.74.3.4	<code>basic_signal_set::basic_signal_set</code> (4 of 4 overloads)	522
5.74.4	<code>basic_signal_set::cancel</code>	522
5.74.4.1	<code>basic_signal_set::cancel</code> (1 of 2 overloads)	522
5.74.4.2	<code>basic_signal_set::cancel</code> (2 of 2 overloads)	523
5.74.5	<code>basic_signal_set::clear</code>	523
5.74.5.1	<code>basic_signal_set::clear</code> (1 of 2 overloads)	523
5.74.5.2	<code>basic_signal_set::clear</code> (2 of 2 overloads)	524
5.74.6	<code>basic_signal_set::get_implementation</code>	524
5.74.6.1	<code>basic_signal_set::get_implementation</code> (1 of 2 overloads)	524
5.74.6.2	<code>basic_signal_set::get_implementation</code> (2 of 2 overloads)	524
5.74.7	<code>basic_signal_set::get_io_service</code>	525
5.74.8	<code>basic_signal_set::get_service</code>	525
5.74.8.1	<code>basic_signal_set::get_service</code> (1 of 2 overloads)	525
5.74.8.2	<code>basic_signal_set::get_service</code> (2 of 2 overloads)	525
5.74.9	<code>basic_signal_set::implementation</code>	525
5.74.10	<code>basic_signal_set::implementation_type</code>	525
5.74.11	<code>basic_signal_set::remove</code>	526
5.74.11.1	<code>basic_signal_set::remove</code> (1 of 2 overloads)	526
5.74.11.2	<code>basic_signal_set::remove</code> (2 of 2 overloads)	526
5.74.12	<code>basic_signal_set::service</code>	527
5.74.13	<code>basic_signal_set::service_type</code>	527
5.75	<code>basic_socket</code>	527
5.75.1	<code>basic_socket::assign</code>	531
5.75.1.1	<code>basic_socket::assign</code> (1 of 2 overloads)	531
5.75.1.2	<code>basic_socket::assign</code> (2 of 2 overloads)	531
5.75.2	<code>basic_socket::async_connect</code>	531
5.75.3	<code>basic_socket::at_mark</code>	532
5.75.3.1	<code>basic_socket::at_mark</code> (1 of 2 overloads)	532

5.75.3.2	<code>basic_socket::at_mark</code> (2 of 2 overloads)	533
5.75.4	<code>basic_socket::available</code>	533
5.75.4.1	<code>basic_socket::available</code> (1 of 2 overloads)	533
5.75.4.2	<code>basic_socket::available</code> (2 of 2 overloads)	533
5.75.5	<code>basic_socket::basic_socket</code>	534
5.75.5.1	<code>basic_socket::basic_socket</code> (1 of 6 overloads)	534
5.75.5.2	<code>basic_socket::basic_socket</code> (2 of 6 overloads)	535
5.75.5.3	<code>basic_socket::basic_socket</code> (3 of 6 overloads)	535
5.75.5.4	<code>basic_socket::basic_socket</code> (4 of 6 overloads)	535
5.75.5.5	<code>basic_socket::basic_socket</code> (5 of 6 overloads)	536
5.75.5.6	<code>basic_socket::basic_socket</code> (6 of 6 overloads)	536
5.75.6	<code>basic_socket::bind</code>	537
5.75.6.1	<code>basic_socket::bind</code> (1 of 2 overloads)	537
5.75.6.2	<code>basic_socket::bind</code> (2 of 2 overloads)	537
5.75.7	<code>basic_socket::broadcast</code>	538
5.75.8	<code>basic_socket::bytes_readable</code>	538
5.75.9	<code>basic_socket::cancel</code>	539
5.75.9.1	<code>basic_socket::cancel</code> (1 of 2 overloads)	539
5.75.9.2	<code>basic_socket::cancel</code> (2 of 2 overloads)	540
5.75.10	<code>basic_socket::close</code>	540
5.75.10.1	<code>basic_socket::close</code> (1 of 2 overloads)	540
5.75.10.2	<code>basic_socket::close</code> (2 of 2 overloads)	541
5.75.11	<code>basic_socket::connect</code>	541
5.75.11.1	<code>basic_socket::connect</code> (1 of 2 overloads)	542
5.75.11.2	<code>basic_socket::connect</code> (2 of 2 overloads)	542
5.75.12	<code>basic_socket::debug</code>	543
5.75.13	<code>basic_socket::do_not_route</code>	543
5.75.14	<code>basic_socket::enable_connection_aborted</code>	544
5.75.15	<code>basic_socket::endpoint_type</code>	544
5.75.16	<code>basic_socket::get_implementation</code>	545
5.75.16.1	<code>basic_socket::get_implementation</code> (1 of 2 overloads)	545
5.75.16.2	<code>basic_socket::get_implementation</code> (2 of 2 overloads)	545
5.75.17	<code>basic_socket::get_io_service</code>	545
5.75.18	<code>basic_socket::get_option</code>	545
5.75.18.1	<code>basic_socket::get_option</code> (1 of 2 overloads)	546
5.75.18.2	<code>basic_socket::get_option</code> (2 of 2 overloads)	546
5.75.19	<code>basic_socket::get_service</code>	547
5.75.19.1	<code>basic_socket::get_service</code> (1 of 2 overloads)	547
5.75.19.2	<code>basic_socket::get_service</code> (2 of 2 overloads)	547

5.75.20	basic_socket::implementation	547
5.75.21	basic_socket::implementation_type	547
5.75.22	basic_socket::io_control	548
5.75.22.1	basic_socket::io_control (1 of 2 overloads)	548
5.75.22.2	basic_socket::io_control (2 of 2 overloads)	549
5.75.23	basic_socket::is_open	549
5.75.24	basic_socket::keep_alive	549
5.75.25	basic_socket::linger	550
5.75.26	basic_socket::local_endpoint	551
5.75.26.1	basic_socket::local_endpoint (1 of 2 overloads)	551
5.75.26.2	basic_socket::local_endpoint (2 of 2 overloads)	551
5.75.27	basic_socket::lowest_layer	552
5.75.27.1	basic_socket::lowest_layer (1 of 2 overloads)	552
5.75.27.2	basic_socket::lowest_layer (2 of 2 overloads)	552
5.75.28	basic_socket::lowest_layer_type	552
5.75.29	basic_socket::max_connections	556
5.75.30	basic_socket::message_do_not_route	556
5.75.31	basic_socket::message_end_of_record	556
5.75.32	basic_socket::message_flags	556
5.75.33	basic_socket::message_out_of_band	557
5.75.34	basic_socket::message_peek	557
5.75.35	basic_socket::native	557
5.75.36	basic_socket::native_handle	557
5.75.37	basic_socket::native_handle_type	557
5.75.38	basic_socket::native_non_blocking	557
5.75.38.1	basic_socket::native_non_blocking (1 of 3 overloads)	558
5.75.38.2	basic_socket::native_non_blocking (2 of 3 overloads)	559
5.75.38.3	basic_socket::native_non_blocking (3 of 3 overloads)	561
5.75.39	basic_socket::native_type	562
5.75.40	basic_socket::non_blocking	563
5.75.40.1	basic_socket::non_blocking (1 of 3 overloads)	563
5.75.40.2	basic_socket::non_blocking (2 of 3 overloads)	563
5.75.40.3	basic_socket::non_blocking (3 of 3 overloads)	564
5.75.41	basic_socket::non_blocking_io	564
5.75.42	basic_socket::open	564
5.75.42.1	basic_socket::open (1 of 2 overloads)	565
5.75.42.2	basic_socket::open (2 of 2 overloads)	565
5.75.43	basic_socket::operator=	566
5.75.43.1	basic_socket::operator= (1 of 2 overloads)	566

5.75.43.2	<code>basic_socket::operator=</code> (2 of 2 overloads)	566
5.75.44	<code>basic_socket::protocol_type</code>	567
5.75.45	<code>basic_socket::receive_buffer_size</code>	567
5.75.46	<code>basic_socket::receive_low_watermark</code>	567
5.75.47	<code>basic_socket::remote_endpoint</code>	568
5.75.47.1	<code>basic_socket::remote_endpoint</code> (1 of 2 overloads)	568
5.75.47.2	<code>basic_socket::remote_endpoint</code> (2 of 2 overloads)	569
5.75.48	<code>basic_socket::reuse_address</code>	569
5.75.49	<code>basic_socket::send_buffer_size</code>	570
5.75.50	<code>basic_socket::send_low_watermark</code>	570
5.75.51	<code>basic_socket::service</code>	571
5.75.52	<code>basic_socket::service_type</code>	571
5.75.53	<code>basic_socket::set_option</code>	571
5.75.53.1	<code>basic_socket::set_option</code> (1 of 2 overloads)	572
5.75.53.2	<code>basic_socket::set_option</code> (2 of 2 overloads)	572
5.75.54	<code>basic_socket::shutdown</code>	573
5.75.54.1	<code>basic_socket::shutdown</code> (1 of 2 overloads)	573
5.75.54.2	<code>basic_socket::shutdown</code> (2 of 2 overloads)	573
5.75.55	<code>basic_socket::shutdown_type</code>	574
5.75.56	<code>basic_socket::~basic_socket</code>	574
5.76	<code>basic_socket_acceptor</code>	574
5.76.1	<code>basic_socket_acceptor::accept</code>	578
5.76.1.1	<code>basic_socket_acceptor::accept</code> (1 of 4 overloads)	578
5.76.1.2	<code>basic_socket_acceptor::accept</code> (2 of 4 overloads)	579
5.76.1.3	<code>basic_socket_acceptor::accept</code> (3 of 4 overloads)	580
5.76.1.4	<code>basic_socket_acceptor::accept</code> (4 of 4 overloads)	580
5.76.2	<code>basic_socket_acceptor::assign</code>	581
5.76.2.1	<code>basic_socket_acceptor::assign</code> (1 of 2 overloads)	581
5.76.2.2	<code>basic_socket_acceptor::assign</code> (2 of 2 overloads)	581
5.76.3	<code>basic_socket_acceptor::async_accept</code>	581
5.76.3.1	<code>basic_socket_acceptor::async_accept</code> (1 of 2 overloads)	582
5.76.3.2	<code>basic_socket_acceptor::async_accept</code> (2 of 2 overloads)	583
5.76.4	<code>basic_socket_acceptor::basic_socket_acceptor</code>	583
5.76.4.1	<code>basic_socket_acceptor::basic_socket_acceptor</code> (1 of 6 overloads)	584
5.76.4.2	<code>basic_socket_acceptor::basic_socket_acceptor</code> (2 of 6 overloads)	584
5.76.4.3	<code>basic_socket_acceptor::basic_socket_acceptor</code> (3 of 6 overloads)	585
5.76.4.4	<code>basic_socket_acceptor::basic_socket_acceptor</code> (4 of 6 overloads)	585
5.76.4.5	<code>basic_socket_acceptor::basic_socket_acceptor</code> (5 of 6 overloads)	586
5.76.4.6	<code>basic_socket_acceptor::basic_socket_acceptor</code> (6 of 6 overloads)	586

5.76.5	<code>basic_socket_acceptor::bind</code>	586
5.76.5.1	<code>basic_socket_acceptor::bind</code> (1 of 2 overloads)	587
5.76.5.2	<code>basic_socket_acceptor::bind</code> (2 of 2 overloads)	587
5.76.6	<code>basic_socket_acceptor::broadcast</code>	588
5.76.7	<code>basic_socket_acceptor::bytes_readable</code>	588
5.76.8	<code>basic_socket_acceptor::cancel</code>	589
5.76.8.1	<code>basic_socket_acceptor::cancel</code> (1 of 2 overloads)	589
5.76.8.2	<code>basic_socket_acceptor::cancel</code> (2 of 2 overloads)	589
5.76.9	<code>basic_socket_acceptor::close</code>	589
5.76.9.1	<code>basic_socket_acceptor::close</code> (1 of 2 overloads)	589
5.76.9.2	<code>basic_socket_acceptor::close</code> (2 of 2 overloads)	590
5.76.10	<code>basic_socket_acceptor::debug</code>	590
5.76.11	<code>basic_socket_acceptor::do_not_route</code>	591
5.76.12	<code>basic_socket_acceptor::enable_connection_aborted</code>	591
5.76.13	<code>basic_socket_acceptor::endpoint_type</code>	592
5.76.14	<code>basic_socket_acceptor::get_implementation</code>	592
5.76.14.1	<code>basic_socket_acceptor::get_implementation</code> (1 of 2 overloads)	592
5.76.14.2	<code>basic_socket_acceptor::get_implementation</code> (2 of 2 overloads)	593
5.76.15	<code>basic_socket_acceptor::get_io_service</code>	593
5.76.16	<code>basic_socket_acceptor::get_option</code>	593
5.76.16.1	<code>basic_socket_acceptor::get_option</code> (1 of 2 overloads)	593
5.76.16.2	<code>basic_socket_acceptor::get_option</code> (2 of 2 overloads)	594
5.76.17	<code>basic_socket_acceptor::get_service</code>	594
5.76.17.1	<code>basic_socket_acceptor::get_service</code> (1 of 2 overloads)	595
5.76.17.2	<code>basic_socket_acceptor::get_service</code> (2 of 2 overloads)	595
5.76.18	<code>basic_socket_acceptor::implementation</code>	595
5.76.19	<code>basic_socket_acceptor::implementation_type</code>	595
5.76.20	<code>basic_socket_acceptor::io_control</code>	595
5.76.20.1	<code>basic_socket_acceptor::io_control</code> (1 of 2 overloads)	596
5.76.20.2	<code>basic_socket_acceptor::io_control</code> (2 of 2 overloads)	596
5.76.21	<code>basic_socket_acceptor::is_open</code>	597
5.76.22	<code>basic_socket_acceptor::keep_alive</code>	597
5.76.23	<code>basic_socket_acceptor::linger</code>	597
5.76.24	<code>basic_socket_acceptor::listen</code>	598
5.76.24.1	<code>basic_socket_acceptor::listen</code> (1 of 2 overloads)	598
5.76.24.2	<code>basic_socket_acceptor::listen</code> (2 of 2 overloads)	599
5.76.25	<code>basic_socket_acceptor::local_endpoint</code>	599
5.76.25.1	<code>basic_socket_acceptor::local_endpoint</code> (1 of 2 overloads)	599
5.76.25.2	<code>basic_socket_acceptor::local_endpoint</code> (2 of 2 overloads)	600

5.76.26	<code>basic_socket_acceptor::max_connections</code>	600
5.76.27	<code>basic_socket_acceptor::message_do_not_route</code>	600
5.76.28	<code>basic_socket_acceptor::message_end_of_record</code>	601
5.76.29	<code>basic_socket_acceptor::message_flags</code>	601
5.76.30	<code>basic_socket_acceptor::message_out_of_band</code>	601
5.76.31	<code>basic_socket_acceptor::message_peek</code>	601
5.76.32	<code>basic_socket_acceptor::native</code>	601
5.76.33	<code>basic_socket_acceptor::native_handle</code>	601
5.76.34	<code>basic_socket_acceptor::native_handle_type</code>	602
5.76.35	<code>basic_socket_acceptor::native_non_blocking</code>	602
5.76.35.1	<code>basic_socket_acceptor::native_non_blocking</code> (1 of 3 overloads)	602
5.76.35.2	<code>basic_socket_acceptor::native_non_blocking</code> (2 of 3 overloads)	602
5.76.35.3	<code>basic_socket_acceptor::native_non_blocking</code> (3 of 3 overloads)	603
5.76.36	<code>basic_socket_acceptor::native_type</code>	603
5.76.37	<code>basic_socket_acceptor::non_blocking</code>	603
5.76.37.1	<code>basic_socket_acceptor::non_blocking</code> (1 of 3 overloads)	604
5.76.37.2	<code>basic_socket_acceptor::non_blocking</code> (2 of 3 overloads)	604
5.76.37.3	<code>basic_socket_acceptor::non_blocking</code> (3 of 3 overloads)	604
5.76.38	<code>basic_socket_acceptor::non_blocking_io</code>	605
5.76.39	<code>basic_socket_acceptor::open</code>	605
5.76.39.1	<code>basic_socket_acceptor::open</code> (1 of 2 overloads)	605
5.76.39.2	<code>basic_socket_acceptor::open</code> (2 of 2 overloads)	606
5.76.40	<code>basic_socket_acceptor::operator=</code>	606
5.76.40.1	<code>basic_socket_acceptor::operator=</code> (1 of 2 overloads)	607
5.76.40.2	<code>basic_socket_acceptor::operator=</code> (2 of 2 overloads)	607
5.76.41	<code>basic_socket_acceptor::protocol_type</code>	607
5.76.42	<code>basic_socket_acceptor::receive_buffer_size</code>	608
5.76.43	<code>basic_socket_acceptor::receive_low_watermark</code>	608
5.76.44	<code>basic_socket_acceptor::reuse_address</code>	609
5.76.45	<code>basic_socket_acceptor::send_buffer_size</code>	609
5.76.46	<code>basic_socket_acceptor::send_low_watermark</code>	610
5.76.47	<code>basic_socket_acceptor::service</code>	610
5.76.48	<code>basic_socket_acceptor::service_type</code>	611
5.76.49	<code>basic_socket_acceptor::set_option</code>	611
5.76.49.1	<code>basic_socket_acceptor::set_option</code> (1 of 2 overloads)	611
5.76.49.2	<code>basic_socket_acceptor::set_option</code> (2 of 2 overloads)	612
5.76.50	<code>basic_socket_acceptor::shutdown_type</code>	612
5.77	<code>basic_socket_iostream</code>	613
5.77.1	<code>basic_socket_iostream::basic_socket_iostream</code>	614

5.77.1.1	<code>basic_socket_iostream::basic_socket_iostream</code> (1 of 2 overloads)	614
5.77.1.2	<code>basic_socket_iostream::basic_socket_iostream</code> (2 of 2 overloads)	614
5.77.2	<code>basic_socket_iostream::close</code>	614
5.77.3	<code>basic_socket_iostream::connect</code>	614
5.77.4	<code>basic_socket_iostream::duration_type</code>	615
5.77.5	<code>basic_socket_iostream::endpoint_type</code>	615
5.77.6	<code>basic_socket_iostream::error</code>	615
5.77.7	<code>basic_socket_iostream::expires_at</code>	616
5.77.7.1	<code>basic_socket_iostream::expires_at</code> (1 of 2 overloads)	616
5.77.7.2	<code>basic_socket_iostream::expires_at</code> (2 of 2 overloads)	616
5.77.8	<code>basic_socket_iostream::expires_from_now</code>	616
5.77.8.1	<code>basic_socket_iostream::expires_from_now</code> (1 of 2 overloads)	616
5.77.8.2	<code>basic_socket_iostream::expires_from_now</code> (2 of 2 overloads)	617
5.77.9	<code>basic_socket_iostream::rdbuf</code>	617
5.77.10	<code>basic_socket_iostream::time_type</code>	617
5.78	<code>basic_socket_streambuf</code>	617
5.78.1	<code>basic_socket_streambuf::assign</code>	621
5.78.1.1	<code>basic_socket_streambuf::assign</code> (1 of 2 overloads)	621
5.78.1.2	<code>basic_socket_streambuf::assign</code> (2 of 2 overloads)	621
5.78.2	<code>basic_socket_streambuf::async_connect</code>	622
5.78.3	<code>basic_socket_streambuf::at_mark</code>	623
5.78.3.1	<code>basic_socket_streambuf::at_mark</code> (1 of 2 overloads)	623
5.78.3.2	<code>basic_socket_streambuf::at_mark</code> (2 of 2 overloads)	623
5.78.4	<code>basic_socket_streambuf::available</code>	623
5.78.4.1	<code>basic_socket_streambuf::available</code> (1 of 2 overloads)	624
5.78.4.2	<code>basic_socket_streambuf::available</code> (2 of 2 overloads)	624
5.78.5	<code>basic_socket_streambuf::basic_socket_streambuf</code>	624
5.78.6	<code>basic_socket_streambuf::bind</code>	624
5.78.6.1	<code>basic_socket_streambuf::bind</code> (1 of 2 overloads)	625
5.78.6.2	<code>basic_socket_streambuf::bind</code> (2 of 2 overloads)	625
5.78.7	<code>basic_socket_streambuf::broadcast</code>	626
5.78.8	<code>basic_socket_streambuf::bytes_readable</code>	626
5.78.9	<code>basic_socket_streambuf::cancel</code>	627
5.78.9.1	<code>basic_socket_streambuf::cancel</code> (1 of 2 overloads)	627
5.78.9.2	<code>basic_socket_streambuf::cancel</code> (2 of 2 overloads)	627
5.78.10	<code>basic_socket_streambuf::close</code>	628
5.78.10.1	<code>basic_socket_streambuf::close</code> (1 of 2 overloads)	628
5.78.10.2	<code>basic_socket_streambuf::close</code> (2 of 2 overloads)	628
5.78.11	<code>basic_socket_streambuf::connect</code>	629

5.78.11.1	basic_socket_streambuf::connect (1 of 3 overloads)	629
5.78.11.2	basic_socket_streambuf::connect (2 of 3 overloads)	630
5.78.11.3	basic_socket_streambuf::connect (3 of 3 overloads)	630
5.78.12	basic_socket_streambuf::debug	631
5.78.13	basic_socket_streambuf::do_not_route	631
5.78.14	basic_socket_streambuf::duration_type	632
5.78.15	basic_socket_streambuf::enable_connection_aborted	632
5.78.16	basic_socket_streambuf::endpoint_type	632
5.78.17	basic_socket_streambuf::error	633
5.78.18	basic_socket_streambuf::expires_at	633
5.78.18.1	basic_socket_streambuf::expires_at (1 of 2 overloads)	633
5.78.18.2	basic_socket_streambuf::expires_at (2 of 2 overloads)	633
5.78.19	basic_socket_streambuf::expires_from_now	634
5.78.19.1	basic_socket_streambuf::expires_from_now (1 of 2 overloads)	634
5.78.19.2	basic_socket_streambuf::expires_from_now (2 of 2 overloads)	634
5.78.20	basic_socket_streambuf::get_implementation	634
5.78.20.1	basic_socket_streambuf::get_implementation (1 of 2 overloads)	634
5.78.20.2	basic_socket_streambuf::get_implementation (2 of 2 overloads)	635
5.78.21	basic_socket_streambuf::get_io_service	635
5.78.22	basic_socket_streambuf::get_option	635
5.78.22.1	basic_socket_streambuf::get_option (1 of 2 overloads)	635
5.78.22.2	basic_socket_streambuf::get_option (2 of 2 overloads)	636
5.78.23	basic_socket_streambuf::get_service	636
5.78.23.1	basic_socket_streambuf::get_service (1 of 2 overloads)	637
5.78.23.2	basic_socket_streambuf::get_service (2 of 2 overloads)	637
5.78.24	basic_socket_streambuf::implementation	637
5.78.25	basic_socket_streambuf::implementation_type	637
5.78.26	basic_socket_streambuf::io_control	637
5.78.26.1	basic_socket_streambuf::io_control (1 of 2 overloads)	638
5.78.26.2	basic_socket_streambuf::io_control (2 of 2 overloads)	638
5.78.27	basic_socket_streambuf::io_handler	639
5.78.28	basic_socket_streambuf::is_open	639
5.78.29	basic_socket_streambuf::keep_alive	639
5.78.30	basic_socket_streambuf::linger	640
5.78.31	basic_socket_streambuf::local_endpoint	640
5.78.31.1	basic_socket_streambuf::local_endpoint (1 of 2 overloads)	640
5.78.31.2	basic_socket_streambuf::local_endpoint (2 of 2 overloads)	641
5.78.32	basic_socket_streambuf::lowest_layer	641
5.78.32.1	basic_socket_streambuf::lowest_layer (1 of 2 overloads)	642

5.78.32.2	<code>basic_socket_streambuf::lowest_layer</code> (2 of 2 overloads)	642
5.78.33	<code>basic_socket_streambuf::lowest_layer_type</code>	642
5.78.34	<code>basic_socket_streambuf::max_connections</code>	646
5.78.35	<code>basic_socket_streambuf::message_do_not_route</code>	646
5.78.36	<code>basic_socket_streambuf::message_end_of_record</code>	646
5.78.37	<code>basic_socket_streambuf::message_flags</code>	646
5.78.38	<code>basic_socket_streambuf::message_out_of_band</code>	646
5.78.39	<code>basic_socket_streambuf::message_peek</code>	647
5.78.40	<code>basic_socket_streambuf::native</code>	647
5.78.41	<code>basic_socket_streambuf::native_handle</code>	647
5.78.42	<code>basic_socket_streambuf::native_handle_type</code>	647
5.78.43	<code>basic_socket_streambuf::native_non_blocking</code>	647
5.78.43.1	<code>basic_socket_streambuf::native_non_blocking</code> (1 of 3 overloads)	648
5.78.43.2	<code>basic_socket_streambuf::native_non_blocking</code> (2 of 3 overloads)	649
5.78.43.3	<code>basic_socket_streambuf::native_non_blocking</code> (3 of 3 overloads)	651
5.78.44	<code>basic_socket_streambuf::native_type</code>	652
5.78.45	<code>basic_socket_streambuf::non_blocking</code>	652
5.78.45.1	<code>basic_socket_streambuf::non_blocking</code> (1 of 3 overloads)	653
5.78.45.2	<code>basic_socket_streambuf::non_blocking</code> (2 of 3 overloads)	653
5.78.45.3	<code>basic_socket_streambuf::non_blocking</code> (3 of 3 overloads)	654
5.78.46	<code>basic_socket_streambuf::non_blocking_io</code>	654
5.78.47	<code>basic_socket_streambuf::open</code>	654
5.78.47.1	<code>basic_socket_streambuf::open</code> (1 of 2 overloads)	655
5.78.47.2	<code>basic_socket_streambuf::open</code> (2 of 2 overloads)	655
5.78.48	<code>basic_socket_streambuf::overflow</code>	656
5.78.49	<code>basic_socket_streambuf::protocol_type</code>	656
5.78.50	<code>basic_socket_streambuf::puberror</code>	656
5.78.51	<code>basic_socket_streambuf::receive_buffer_size</code>	656
5.78.52	<code>basic_socket_streambuf::receive_low_watermark</code>	657
5.78.53	<code>basic_socket_streambuf::remote_endpoint</code>	657
5.78.53.1	<code>basic_socket_streambuf::remote_endpoint</code> (1 of 2 overloads)	657
5.78.53.2	<code>basic_socket_streambuf::remote_endpoint</code> (2 of 2 overloads)	658
5.78.54	<code>basic_socket_streambuf::reuse_address</code>	658
5.78.55	<code>basic_socket_streambuf::send_buffer_size</code>	659
5.78.56	<code>basic_socket_streambuf::send_low_watermark</code>	660
5.78.57	<code>basic_socket_streambuf::service</code>	660
5.78.58	<code>basic_socket_streambuf::service_type</code>	660
5.78.59	<code>basic_socket_streambuf::set_option</code>	661
5.78.59.1	<code>basic_socket_streambuf::set_option</code> (1 of 2 overloads)	661

5.78.59.2	<code>basic_socket_streambuf::set_option</code> (2 of 2 overloads)	661
5.78.60	<code>basic_socket_streambuf::setbuf</code>	662
5.78.61	<code>basic_socket_streambuf::shutdown</code>	662
5.78.61.1	<code>basic_socket_streambuf::shutdown</code> (1 of 2 overloads)	662
5.78.61.2	<code>basic_socket_streambuf::shutdown</code> (2 of 2 overloads)	663
5.78.62	<code>basic_socket_streambuf::shutdown_type</code>	663
5.78.63	<code>basic_socket_streambuf::sync</code>	664
5.78.64	<code>basic_socket_streambuf::time_type</code>	664
5.78.65	<code>basic_socket_streambuf::timer_handler</code>	664
5.78.66	<code>basic_socket_streambuf::underflow</code>	664
5.78.67	<code>basic_socket_streambuf::~basic_socket_streambuf</code>	664
5.79	<code>basic_stream_socket</code>	664
5.79.1	<code>basic_stream_socket::assign</code>	668
5.79.1.1	<code>basic_stream_socket::assign</code> (1 of 2 overloads)	668
5.79.1.2	<code>basic_stream_socket::assign</code> (2 of 2 overloads)	669
5.79.2	<code>basic_stream_socket::async_connect</code>	669
5.79.3	<code>basic_stream_socket::async_read_some</code>	670
5.79.4	<code>basic_stream_socket::async_receive</code>	670
5.79.4.1	<code>basic_stream_socket::async_receive</code> (1 of 2 overloads)	671
5.79.4.2	<code>basic_stream_socket::async_receive</code> (2 of 2 overloads)	672
5.79.5	<code>basic_stream_socket::async_send</code>	672
5.79.5.1	<code>basic_stream_socket::async_send</code> (1 of 2 overloads)	673
5.79.5.2	<code>basic_stream_socket::async_send</code> (2 of 2 overloads)	674
5.79.6	<code>basic_stream_socket::async_write_some</code>	674
5.79.7	<code>basic_stream_socket::at_mark</code>	675
5.79.7.1	<code>basic_stream_socket::at_mark</code> (1 of 2 overloads)	675
5.79.7.2	<code>basic_stream_socket::at_mark</code> (2 of 2 overloads)	676
5.79.8	<code>basic_stream_socket::available</code>	676
5.79.8.1	<code>basic_stream_socket::available</code> (1 of 2 overloads)	676
5.79.8.2	<code>basic_stream_socket::available</code> (2 of 2 overloads)	677
5.79.9	<code>basic_stream_socket::basic_stream_socket</code>	677
5.79.9.1	<code>basic_stream_socket::basic_stream_socket</code> (1 of 6 overloads)	678
5.79.9.2	<code>basic_stream_socket::basic_stream_socket</code> (2 of 6 overloads)	678
5.79.9.3	<code>basic_stream_socket::basic_stream_socket</code> (3 of 6 overloads)	678
5.79.9.4	<code>basic_stream_socket::basic_stream_socket</code> (4 of 6 overloads)	679
5.79.9.5	<code>basic_stream_socket::basic_stream_socket</code> (5 of 6 overloads)	679
5.79.9.6	<code>basic_stream_socket::basic_stream_socket</code> (6 of 6 overloads)	680
5.79.10	<code>basic_stream_socket::bind</code>	680
5.79.10.1	<code>basic_stream_socket::bind</code> (1 of 2 overloads)	680

5.79.10.2	<code>basic_stream_socket::bind</code> (2 of 2 overloads)	681
5.79.11	<code>basic_stream_socket::broadcast</code>	681
5.79.12	<code>basic_stream_socket::bytes_readable</code>	682
5.79.13	<code>basic_stream_socket::cancel</code>	682
5.79.13.1	<code>basic_stream_socket::cancel</code> (1 of 2 overloads)	682
5.79.13.2	<code>basic_stream_socket::cancel</code> (2 of 2 overloads)	683
5.79.14	<code>basic_stream_socket::close</code>	684
5.79.14.1	<code>basic_stream_socket::close</code> (1 of 2 overloads)	684
5.79.14.2	<code>basic_stream_socket::close</code> (2 of 2 overloads)	684
5.79.15	<code>basic_stream_socket::connect</code>	685
5.79.15.1	<code>basic_stream_socket::connect</code> (1 of 2 overloads)	685
5.79.15.2	<code>basic_stream_socket::connect</code> (2 of 2 overloads)	686
5.79.16	<code>basic_stream_socket::debug</code>	686
5.79.17	<code>basic_stream_socket::do_not_route</code>	687
5.79.18	<code>basic_stream_socket::enable_connection_aborted</code>	687
5.79.19	<code>basic_stream_socket::endpoint_type</code>	688
5.79.20	<code>basic_stream_socket::get_implementation</code>	688
5.79.20.1	<code>basic_stream_socket::get_implementation</code> (1 of 2 overloads)	688
5.79.20.2	<code>basic_stream_socket::get_implementation</code> (2 of 2 overloads)	689
5.79.21	<code>basic_stream_socket::get_io_service</code>	689
5.79.22	<code>basic_stream_socket::get_option</code>	689
5.79.22.1	<code>basic_stream_socket::get_option</code> (1 of 2 overloads)	689
5.79.22.2	<code>basic_stream_socket::get_option</code> (2 of 2 overloads)	690
5.79.23	<code>basic_stream_socket::get_service</code>	690
5.79.23.1	<code>basic_stream_socket::get_service</code> (1 of 2 overloads)	691
5.79.23.2	<code>basic_stream_socket::get_service</code> (2 of 2 overloads)	691
5.79.24	<code>basic_stream_socket::implementation</code>	691
5.79.25	<code>basic_stream_socket::implementation_type</code>	691
5.79.26	<code>basic_stream_socket::io_control</code>	691
5.79.26.1	<code>basic_stream_socket::io_control</code> (1 of 2 overloads)	692
5.79.26.2	<code>basic_stream_socket::io_control</code> (2 of 2 overloads)	692
5.79.27	<code>basic_stream_socket::is_open</code>	693
5.79.28	<code>basic_stream_socket::keep_alive</code>	693
5.79.29	<code>basic_stream_socket::linger</code>	694
5.79.30	<code>basic_stream_socket::local_endpoint</code>	694
5.79.30.1	<code>basic_stream_socket::local_endpoint</code> (1 of 2 overloads)	694
5.79.30.2	<code>basic_stream_socket::local_endpoint</code> (2 of 2 overloads)	695
5.79.31	<code>basic_stream_socket::lowest_layer</code>	695
5.79.31.1	<code>basic_stream_socket::lowest_layer</code> (1 of 2 overloads)	696

5.79.31.2	<code>basic_stream_socket::lowest_layer</code> (2 of 2 overloads)	696
5.79.32	<code>basic_stream_socket::lowest_layer_type</code>	696
5.79.33	<code>basic_stream_socket::max_connections</code>	700
5.79.34	<code>basic_stream_socket::message_do_not_route</code>	700
5.79.35	<code>basic_stream_socket::message_end_of_record</code>	700
5.79.36	<code>basic_stream_socket::message_flags</code>	700
5.79.37	<code>basic_stream_socket::message_out_of_band</code>	700
5.79.38	<code>basic_stream_socket::message_peek</code>	701
5.79.39	<code>basic_stream_socket::native</code>	701
5.79.40	<code>basic_stream_socket::native_handle</code>	701
5.79.41	<code>basic_stream_socket::native_handle_type</code>	701
5.79.42	<code>basic_stream_socket::native_non_blocking</code>	701
5.79.42.1	<code>basic_stream_socket::native_non_blocking</code> (1 of 3 overloads)	702
5.79.42.2	<code>basic_stream_socket::native_non_blocking</code> (2 of 3 overloads)	703
5.79.42.3	<code>basic_stream_socket::native_non_blocking</code> (3 of 3 overloads)	705
5.79.43	<code>basic_stream_socket::native_type</code>	706
5.79.44	<code>basic_stream_socket::non_blocking</code>	706
5.79.44.1	<code>basic_stream_socket::non_blocking</code> (1 of 3 overloads)	707
5.79.44.2	<code>basic_stream_socket::non_blocking</code> (2 of 3 overloads)	707
5.79.44.3	<code>basic_stream_socket::non_blocking</code> (3 of 3 overloads)	708
5.79.45	<code>basic_stream_socket::non_blocking_io</code>	708
5.79.46	<code>basic_stream_socket::open</code>	708
5.79.46.1	<code>basic_stream_socket::open</code> (1 of 2 overloads)	709
5.79.46.2	<code>basic_stream_socket::open</code> (2 of 2 overloads)	709
5.79.47	<code>basic_stream_socket::operator=</code>	710
5.79.47.1	<code>basic_stream_socket::operator=</code> (1 of 2 overloads)	710
5.79.47.2	<code>basic_stream_socket::operator=</code> (2 of 2 overloads)	710
5.79.48	<code>basic_stream_socket::protocol_type</code>	711
5.79.49	<code>basic_stream_socket::read_some</code>	711
5.79.49.1	<code>basic_stream_socket::read_some</code> (1 of 2 overloads)	711
5.79.49.2	<code>basic_stream_socket::read_some</code> (2 of 2 overloads)	712
5.79.50	<code>basic_stream_socket::receive</code>	712
5.79.50.1	<code>basic_stream_socket::receive</code> (1 of 3 overloads)	713
5.79.50.2	<code>basic_stream_socket::receive</code> (2 of 3 overloads)	714
5.79.50.3	<code>basic_stream_socket::receive</code> (3 of 3 overloads)	714
5.79.51	<code>basic_stream_socket::receive_buffer_size</code>	715
5.79.52	<code>basic_stream_socket::receive_low_watermark</code>	715
5.79.53	<code>basic_stream_socket::remote_endpoint</code>	716
5.79.53.1	<code>basic_stream_socket::remote_endpoint</code> (1 of 2 overloads)	716

5.79.53.2	<code>basic_stream_socket::remote_endpoint</code> (2 of 2 overloads)	717
5.79.54	<code>basic_stream_socket::reuse_address</code>	717
5.79.55	<code>basic_stream_socket::send</code>	718
5.79.55.1	<code>basic_stream_socket::send</code> (1 of 3 overloads)	718
5.79.55.2	<code>basic_stream_socket::send</code> (2 of 3 overloads)	719
5.79.55.3	<code>basic_stream_socket::send</code> (3 of 3 overloads)	720
5.79.56	<code>basic_stream_socket::send_buffer_size</code>	720
5.79.57	<code>basic_stream_socket::send_low_watermark</code>	721
5.79.58	<code>basic_stream_socket::service</code>	721
5.79.59	<code>basic_stream_socket::service_type</code>	721
5.79.60	<code>basic_stream_socket::set_option</code>	722
5.79.60.1	<code>basic_stream_socket::set_option</code> (1 of 2 overloads)	722
5.79.60.2	<code>basic_stream_socket::set_option</code> (2 of 2 overloads)	723
5.79.61	<code>basic_stream_socket::shutdown</code>	723
5.79.61.1	<code>basic_stream_socket::shutdown</code> (1 of 2 overloads)	723
5.79.61.2	<code>basic_stream_socket::shutdown</code> (2 of 2 overloads)	724
5.79.62	<code>basic_stream_socket::shutdown_type</code>	724
5.79.63	<code>basic_stream_socket::write_some</code>	725
5.79.63.1	<code>basic_stream_socket::write_some</code> (1 of 2 overloads)	725
5.79.63.2	<code>basic_stream_socket::write_some</code> (2 of 2 overloads)	726
5.80	<code>basic_streampbuf</code>	726
5.80.1	<code>basic_streampbuf::basic_streampbuf</code>	728
5.80.2	<code>basic_streampbuf::commit</code>	728
5.80.3	<code>basic_streampbuf::const_buffers_type</code>	729
5.80.4	<code>basic_streampbuf::consume</code>	729
5.80.5	<code>basic_streampbuf::data</code>	729
5.80.6	<code>basic_streampbuf::max_size</code>	729
5.80.7	<code>basic_streampbuf::mutable_buffers_type</code>	730
5.80.8	<code>basic_streampbuf::overflow</code>	730
5.80.9	<code>basic_streampbuf::prepare</code>	730
5.80.10	<code>basic_streampbuf::reserve</code>	730
5.80.11	<code>basic_streampbuf::size</code>	731
5.80.12	<code>basic_streampbuf::underflow</code>	731
5.81	<code>basic_waitable_timer</code>	731
5.81.1	<code>basic_waitable_timer::async_wait</code>	734
5.81.2	<code>basic_waitable_timer::basic_waitable_timer</code>	735
5.81.2.1	<code>basic_waitable_timer::basic_waitable_timer</code> (1 of 3 overloads)	735
5.81.2.2	<code>basic_waitable_timer::basic_waitable_timer</code> (2 of 3 overloads)	735
5.81.2.3	<code>basic_waitable_timer::basic_waitable_timer</code> (3 of 3 overloads)	735

5.81.3	<code>basic_waitable_timer::cancel</code>	736
5.81.3.1	<code>basic_waitable_timer::cancel</code> (1 of 2 overloads)	736
5.81.3.2	<code>basic_waitable_timer::cancel</code> (2 of 2 overloads)	736
5.81.4	<code>basic_waitable_timer::cancel_one</code>	737
5.81.4.1	<code>basic_waitable_timer::cancel_one</code> (1 of 2 overloads)	737
5.81.4.2	<code>basic_waitable_timer::cancel_one</code> (2 of 2 overloads)	738
5.81.5	<code>basic_waitable_timer::clock_type</code>	738
5.81.6	<code>basic_waitable_timer::duration</code>	739
5.81.7	<code>basic_waitable_timer::expires_at</code>	739
5.81.7.1	<code>basic_waitable_timer::expires_at</code> (1 of 3 overloads)	739
5.81.7.2	<code>basic_waitable_timer::expires_at</code> (2 of 3 overloads)	739
5.81.7.3	<code>basic_waitable_timer::expires_at</code> (3 of 3 overloads)	740
5.81.8	<code>basic_waitable_timer::expires_from_now</code>	741
5.81.8.1	<code>basic_waitable_timer::expires_from_now</code> (1 of 3 overloads)	741
5.81.8.2	<code>basic_waitable_timer::expires_from_now</code> (2 of 3 overloads)	741
5.81.8.3	<code>basic_waitable_timer::expires_from_now</code> (3 of 3 overloads)	742
5.81.9	<code>basic_waitable_timer::get_implementation</code>	742
5.81.9.1	<code>basic_waitable_timer::get_implementation</code> (1 of 2 overloads)	742
5.81.9.2	<code>basic_waitable_timer::get_implementation</code> (2 of 2 overloads)	742
5.81.10	<code>basic_waitable_timer::get_io_service</code>	743
5.81.11	<code>basic_waitable_timer::get_service</code>	743
5.81.11.1	<code>basic_waitable_timer::get_service</code> (1 of 2 overloads)	743
5.81.11.2	<code>basic_waitable_timer::get_service</code> (2 of 2 overloads)	743
5.81.12	<code>basic_waitable_timer::implementation</code>	743
5.81.13	<code>basic_waitable_timer::implementation_type</code>	743
5.81.14	<code>basic_waitable_timer::service</code>	744
5.81.15	<code>basic_waitable_timer::service_type</code>	744
5.81.16	<code>basic_waitable_timer::time_point</code>	744
5.81.17	<code>basic_waitable_timer::traits_type</code>	744
5.81.18	<code>basic_waitable_timer::wait</code>	745
5.81.18.1	<code>basic_waitable_timer::wait</code> (1 of 2 overloads)	745
5.81.18.2	<code>basic_waitable_timer::wait</code> (2 of 2 overloads)	745
5.82	<code>basic_yield_context</code>	745
5.82.1	<code>basic_yield_context::basic_yield_context</code>	746
5.82.2	<code>basic_yield_context::callee_type</code>	746
5.82.3	<code>basic_yield_context::caller_type</code>	747
5.82.4	<code>basic_yield_context::operator[]</code>	747
5.83	<code>buffer</code>	747
5.83.1	<code>buffer</code> (1 of 28 overloads)	753

5.83.2	buffer (2 of 28 overloads)	753
5.83.3	buffer (3 of 28 overloads)	754
5.83.4	buffer (4 of 28 overloads)	754
5.83.5	buffer (5 of 28 overloads)	754
5.83.6	buffer (6 of 28 overloads)	754
5.83.7	buffer (7 of 28 overloads)	755
5.83.8	buffer (8 of 28 overloads)	755
5.83.9	buffer (9 of 28 overloads)	755
5.83.10	buffer (10 of 28 overloads)	756
5.83.11	buffer (11 of 28 overloads)	756
5.83.12	buffer (12 of 28 overloads)	756
5.83.13	buffer (13 of 28 overloads)	757
5.83.14	buffer (14 of 28 overloads)	757
5.83.15	buffer (15 of 28 overloads)	757
5.83.16	buffer (16 of 28 overloads)	758
5.83.17	buffer (17 of 28 overloads)	758
5.83.18	buffer (18 of 28 overloads)	758
5.83.19	buffer (19 of 28 overloads)	759
5.83.20	buffer (20 of 28 overloads)	759
5.83.21	buffer (21 of 28 overloads)	759
5.83.22	buffer (22 of 28 overloads)	760
5.83.23	buffer (23 of 28 overloads)	760
5.83.24	buffer (24 of 28 overloads)	760
5.83.25	buffer (25 of 28 overloads)	761
5.83.26	buffer (26 of 28 overloads)	761
5.83.27	buffer (27 of 28 overloads)	762
5.83.28	buffer (28 of 28 overloads)	762
5.84	buffer_cast	763
5.84.1	buffer_cast (1 of 2 overloads)	763
5.84.2	buffer_cast (2 of 2 overloads)	763
5.85	buffer_copy	764
5.85.1	buffer_copy (1 of 30 overloads)	767
5.85.2	buffer_copy (2 of 30 overloads)	768
5.85.3	buffer_copy (3 of 30 overloads)	768
5.85.4	buffer_copy (4 of 30 overloads)	769
5.85.5	buffer_copy (5 of 30 overloads)	769
5.85.6	buffer_copy (6 of 30 overloads)	770
5.85.7	buffer_copy (7 of 30 overloads)	770
5.85.8	buffer_copy (8 of 30 overloads)	771

5.85.9	buffer_copy (9 of 30 overloads)	772
5.85.10	buffer_copy (10 of 30 overloads)	772
5.85.11	buffer_copy (11 of 30 overloads)	773
5.85.12	buffer_copy (12 of 30 overloads)	773
5.85.13	buffer_copy (13 of 30 overloads)	774
5.85.14	buffer_copy (14 of 30 overloads)	774
5.85.15	buffer_copy (15 of 30 overloads)	775
5.85.16	buffer_copy (16 of 30 overloads)	776
5.85.17	buffer_copy (17 of 30 overloads)	776
5.85.18	buffer_copy (18 of 30 overloads)	777
5.85.19	buffer_copy (19 of 30 overloads)	777
5.85.20	buffer_copy (20 of 30 overloads)	778
5.85.21	buffer_copy (21 of 30 overloads)	779
5.85.22	buffer_copy (22 of 30 overloads)	779
5.85.23	buffer_copy (23 of 30 overloads)	780
5.85.24	buffer_copy (24 of 30 overloads)	780
5.85.25	buffer_copy (25 of 30 overloads)	781
5.85.26	buffer_copy (26 of 30 overloads)	782
5.85.27	buffer_copy (27 of 30 overloads)	782
5.85.28	buffer_copy (28 of 30 overloads)	783
5.85.29	buffer_copy (29 of 30 overloads)	784
5.85.30	buffer_copy (30 of 30 overloads)	784
5.86	buffer_size	785
5.86.1	buffer_size (1 of 5 overloads)	785
5.86.2	buffer_size (2 of 5 overloads)	786
5.86.3	buffer_size (3 of 5 overloads)	786
5.86.4	buffer_size (4 of 5 overloads)	786
5.86.5	buffer_size (5 of 5 overloads)	786
5.87	buffered_read_stream	786
5.87.1	buffered_read_stream::async_fill	788
5.87.2	buffered_read_stream::async_read_some	788
5.87.3	buffered_read_stream::async_write_some	788
5.87.4	buffered_read_stream::buffered_read_stream	789
5.87.4.1	buffered_read_stream::buffered_read_stream (1 of 2 overloads)	789
5.87.4.2	buffered_read_stream::buffered_read_stream (2 of 2 overloads)	789
5.87.5	buffered_read_stream::close	789
5.87.5.1	buffered_read_stream::close (1 of 2 overloads)	789
5.87.5.2	buffered_read_stream::close (2 of 2 overloads)	789
5.87.6	buffered_read_stream::default_buffer_size	790

5.87.7	buffered_read_stream::fill	790
5.87.7.1	buffered_read_stream::fill (1 of 2 overloads)	790
5.87.7.2	buffered_read_stream::fill (2 of 2 overloads)	790
5.87.8	buffered_read_stream::get_io_service	790
5.87.9	buffered_read_stream::in_avail	790
5.87.9.1	buffered_read_stream::in_avail (1 of 2 overloads)	790
5.87.9.2	buffered_read_stream::in_avail (2 of 2 overloads)	791
5.87.10	buffered_read_stream::lowest_layer	791
5.87.10.1	buffered_read_stream::lowest_layer (1 of 2 overloads)	791
5.87.10.2	buffered_read_stream::lowest_layer (2 of 2 overloads)	791
5.87.11	buffered_read_stream::lowest_layer_type	791
5.87.12	buffered_read_stream::next_layer	791
5.87.13	buffered_read_stream::next_layer_type	791
5.87.14	buffered_read_stream::peek	792
5.87.14.1	buffered_read_stream::peek (1 of 2 overloads)	792
5.87.14.2	buffered_read_stream::peek (2 of 2 overloads)	792
5.87.15	buffered_read_stream::read_some	792
5.87.15.1	buffered_read_stream::read_some (1 of 2 overloads)	793
5.87.15.2	buffered_read_stream::read_some (2 of 2 overloads)	793
5.87.16	buffered_read_stream::write_some	793
5.87.16.1	buffered_read_stream::write_some (1 of 2 overloads)	793
5.87.16.2	buffered_read_stream::write_some (2 of 2 overloads)	794
5.88	buffered_stream	794
5.88.1	buffered_stream::async_fill	795
5.88.2	buffered_stream::async_flush	796
5.88.3	buffered_stream::async_read_some	796
5.88.4	buffered_stream::async_write_some	796
5.88.5	buffered_stream::buffered_stream	796
5.88.5.1	buffered_stream::buffered_stream (1 of 2 overloads)	797
5.88.5.2	buffered_stream::buffered_stream (2 of 2 overloads)	797
5.88.6	buffered_stream::close	797
5.88.6.1	buffered_stream::close (1 of 2 overloads)	797
5.88.6.2	buffered_stream::close (2 of 2 overloads)	797
5.88.7	buffered_stream::fill	797
5.88.7.1	buffered_stream::fill (1 of 2 overloads)	798
5.88.7.2	buffered_stream::fill (2 of 2 overloads)	798
5.88.8	buffered_stream::flush	798
5.88.8.1	buffered_stream::flush (1 of 2 overloads)	798
5.88.8.2	buffered_stream::flush (2 of 2 overloads)	798

5.88.9	buffered_stream::get_io_service	798
5.88.10	buffered_stream::in_avail	799
5.88.10.1	buffered_stream::in_avail (1 of 2 overloads)	799
5.88.10.2	buffered_stream::in_avail (2 of 2 overloads)	799
5.88.11	buffered_stream::lowest_layer	799
5.88.11.1	buffered_stream::lowest_layer (1 of 2 overloads)	799
5.88.11.2	buffered_stream::lowest_layer (2 of 2 overloads)	799
5.88.12	buffered_stream::lowest_layer_type	799
5.88.13	buffered_stream::next_layer	800
5.88.14	buffered_stream::next_layer_type	800
5.88.15	buffered_stream::peek	800
5.88.15.1	buffered_stream::peek (1 of 2 overloads)	800
5.88.15.2	buffered_stream::peek (2 of 2 overloads)	801
5.88.16	buffered_stream::read_some	801
5.88.16.1	buffered_stream::read_some (1 of 2 overloads)	801
5.88.16.2	buffered_stream::read_some (2 of 2 overloads)	801
5.88.17	buffered_stream::write_some	801
5.88.17.1	buffered_stream::write_some (1 of 2 overloads)	802
5.88.17.2	buffered_stream::write_some (2 of 2 overloads)	802
5.89	buffered_write_stream	802
5.89.1	buffered_write_stream::async_flush	804
5.89.2	buffered_write_stream::async_read_some	804
5.89.3	buffered_write_stream::async_write_some	804
5.89.4	buffered_write_stream::buffered_write_stream	804
5.89.4.1	buffered_write_stream::buffered_write_stream (1 of 2 overloads)	805
5.89.4.2	buffered_write_stream::buffered_write_stream (2 of 2 overloads)	805
5.89.5	buffered_write_stream::close	805
5.89.5.1	buffered_write_stream::close (1 of 2 overloads)	805
5.89.5.2	buffered_write_stream::close (2 of 2 overloads)	805
5.89.6	buffered_write_stream::default_buffer_size	805
5.89.7	buffered_write_stream::flush	806
5.89.7.1	buffered_write_stream::flush (1 of 2 overloads)	806
5.89.7.2	buffered_write_stream::flush (2 of 2 overloads)	806
5.89.8	buffered_write_stream::get_io_service	806
5.89.9	buffered_write_stream::in_avail	806
5.89.9.1	buffered_write_stream::in_avail (1 of 2 overloads)	806
5.89.9.2	buffered_write_stream::in_avail (2 of 2 overloads)	807
5.89.10	buffered_write_stream::lowest_layer	807
5.89.10.1	buffered_write_stream::lowest_layer (1 of 2 overloads)	807

5.89.10.2	buffered_write_stream::lowest_layer (2 of 2 overloads)	807
5.89.11	buffered_write_stream::lowest_layer_type	807
5.89.12	buffered_write_stream::next_layer	807
5.89.13	buffered_write_stream::next_layer_type	807
5.89.14	buffered_write_stream::peek	808
5.89.14.1	buffered_write_stream::peek (1 of 2 overloads)	808
5.89.14.2	buffered_write_stream::peek (2 of 2 overloads)	808
5.89.15	buffered_write_stream::read_some	808
5.89.15.1	buffered_write_stream::read_some (1 of 2 overloads)	809
5.89.15.2	buffered_write_stream::read_some (2 of 2 overloads)	809
5.89.16	buffered_write_stream::write_some	809
5.89.16.1	buffered_write_stream::write_some (1 of 2 overloads)	809
5.89.16.2	buffered_write_stream::write_some (2 of 2 overloads)	810
5.90	buffers_begin	810
5.91	buffers_end	810
5.92	buffers_iterator	810
5.92.1	buffers_iterator::begin	812
5.92.2	buffers_iterator::buffers_iterator	812
5.92.3	buffers_iterator::difference_type	812
5.92.4	buffers_iterator::end	812
5.92.5	buffers_iterator::iterator_category	812
5.92.6	buffers_iterator::operator *	813
5.92.7	buffers_iterator::operator!=	813
5.92.8	buffers_iterator::operator+	813
5.92.8.1	buffers_iterator::operator+ (1 of 2 overloads)	813
5.92.8.2	buffers_iterator::operator+ (2 of 2 overloads)	814
5.92.9	buffers_iterator::operator++	814
5.92.9.1	buffers_iterator::operator++ (1 of 2 overloads)	814
5.92.9.2	buffers_iterator::operator++ (2 of 2 overloads)	814
5.92.10	buffers_iterator::operator+=	814
5.92.11	buffers_iterator::operator-	814
5.92.11.1	buffers_iterator::operator- (1 of 2 overloads)	815
5.92.11.2	buffers_iterator::operator- (2 of 2 overloads)	815
5.92.12	buffers_iterator::operator--	815
5.92.12.1	buffers_iterator::operator-- (1 of 2 overloads)	815
5.92.12.2	buffers_iterator::operator-- (2 of 2 overloads)	815
5.92.13	buffers_iterator::operator-=	816
5.92.14	buffers_iterator::operator->	816
5.92.15	buffers_iterator::operator<	816

5.92.16	buffers_iterator::operator<=	816
5.92.17	buffers_iterator::operator==	816
5.92.18	buffers_iterator::operator>	817
5.92.19	buffers_iterator::operator>=	817
5.92.20	buffers_iterator::operator[]	817
5.92.21	buffers_iterator::pointer	817
5.92.22	buffers_iterator::reference	817
5.92.23	buffers_iterator::value_type	818
5.93	connect	818
5.93.1	connect (1 of 8 overloads)	819
5.93.2	connect (2 of 8 overloads)	820
5.93.3	connect (3 of 8 overloads)	821
5.93.4	connect (4 of 8 overloads)	822
5.93.5	connect (5 of 8 overloads)	823
5.93.6	connect (6 of 8 overloads)	824
5.93.7	connect (7 of 8 overloads)	826
5.93.8	connect (8 of 8 overloads)	827
5.94	const_buffer	828
5.94.1	const_buffer::const_buffer	829
5.94.1.1	const_buffer::const_buffer (1 of 3 overloads)	829
5.94.1.2	const_buffer::const_buffer (2 of 3 overloads)	829
5.94.1.3	const_buffer::const_buffer (3 of 3 overloads)	830
5.94.2	const_buffer::operator+	830
5.94.2.1	const_buffer::operator+ (1 of 2 overloads)	830
5.94.2.2	const_buffer::operator+ (2 of 2 overloads)	830
5.95	const_buffers_1	830
5.95.1	const_buffers_1::begin	831
5.95.2	const_buffers_1::const_buffers_1	831
5.95.2.1	const_buffers_1::const_buffers_1 (1 of 2 overloads)	831
5.95.2.2	const_buffers_1::const_buffers_1 (2 of 2 overloads)	832
5.95.3	const_buffers_1::const_iterator	832
5.95.4	const_buffers_1::end	832
5.95.5	const_buffers_1::operator+	832
5.95.5.1	const_buffers_1::operator+ (1 of 2 overloads)	832
5.95.5.2	const_buffers_1::operator+ (2 of 2 overloads)	832
5.95.6	const_buffers_1::value_type	833
5.96	coroutine	833
5.96.1	coroutine::coroutine	837
5.96.2	coroutine::is_child	837

5.96.3	coroutine::is_complete	838
5.96.4	coroutine::is_parent	838
5.97	datagram_socket_service	838
5.97.1	datagram_socket_service::assign	840
5.97.2	datagram_socket_service::async_connect	840
5.97.3	datagram_socket_service::async_receive	840
5.97.4	datagram_socket_service::async_receive_from	841
5.97.5	datagram_socket_service::async_send	841
5.97.6	datagram_socket_service::async_send_to	841
5.97.7	datagram_socket_service::at_mark	841
5.97.8	datagram_socket_service::available	842
5.97.9	datagram_socket_service::bind	842
5.97.10	datagram_socket_service::cancel	842
5.97.11	datagram_socket_service::close	842
5.97.12	datagram_socket_service::connect	842
5.97.13	datagram_socket_service::construct	842
5.97.14	datagram_socket_service::converting_move_construct	843
5.97.15	datagram_socket_service::datagram_socket_service	843
5.97.16	datagram_socket_service::destroy	843
5.97.17	datagram_socket_service::endpoint_type	843
5.97.18	datagram_socket_service::get_io_service	843
5.97.19	datagram_socket_service::get_option	843
5.97.20	datagram_socket_service::id	844
5.97.21	datagram_socket_service::implementation_type	844
5.97.22	datagram_socket_service::io_control	844
5.97.23	datagram_socket_service::is_open	844
5.97.24	datagram_socket_service::local_endpoint	844
5.97.25	datagram_socket_service::move_assign	844
5.97.26	datagram_socket_service::move_construct	845
5.97.27	datagram_socket_service::native	845
5.97.28	datagram_socket_service::native_handle	845
5.97.29	datagram_socket_service::native_handle_type	845
5.97.30	datagram_socket_service::native_non_blocking	845
5.97.30.1	datagram_socket_service::native_non_blocking (1 of 2 overloads)	845
5.97.30.2	datagram_socket_service::native_non_blocking (2 of 2 overloads)	846
5.97.31	datagram_socket_service::native_type	846
5.97.32	datagram_socket_service::non_blocking	846
5.97.32.1	datagram_socket_service::non_blocking (1 of 2 overloads)	846
5.97.32.2	datagram_socket_service::non_blocking (2 of 2 overloads)	846

5.97.33	datagram_socket_service::open	847
5.97.34	datagram_socket_service::protocol_type	847
5.97.35	datagram_socket_service::receive	847
5.97.36	datagram_socket_service::receive_from	847
5.97.37	datagram_socket_service::remote_endpoint	847
5.97.38	datagram_socket_service::send	848
5.97.39	datagram_socket_service::send_to	848
5.97.40	datagram_socket_service::set_option	848
5.97.41	datagram_socket_service::shutdown	848
5.98	deadline_timer	848
5.99	deadline_timer_service	851
5.99.1	deadline_timer_service::async_wait	852
5.99.2	deadline_timer_service::cancel	852
5.99.3	deadline_timer_service::cancel_one	853
5.99.4	deadline_timer_service::construct	853
5.99.5	deadline_timer_service::deadline_timer_service	853
5.99.6	deadline_timer_service::destroy	853
5.99.7	deadline_timer_service::duration_type	853
5.99.8	deadline_timer_service::expires_at	854
5.99.8.1	deadline_timer_service::expires_at (1 of 2 overloads)	854
5.99.8.2	deadline_timer_service::expires_at (2 of 2 overloads)	854
5.99.9	deadline_timer_service::expires_from_now	854
5.99.9.1	deadline_timer_service::expires_from_now (1 of 2 overloads)	854
5.99.9.2	deadline_timer_service::expires_from_now (2 of 2 overloads)	855
5.99.10	deadline_timer_service::get_io_service	855
5.99.11	deadline_timer_service::id	855
5.99.12	deadline_timer_service::implementation_type	855
5.99.13	deadline_timer_service::time_type	855
5.99.14	deadline_timer_service::traits_type	855
5.99.15	deadline_timer_service::wait	856
5.100	error::addrinfo_category	856
5.101	error::addrinfo_errors	856
5.102	error::basic_errors	856
5.103	error::get_addrinfo_category	858
5.104	error::get_misc_category	858
5.105	error::get_netdb_category	858
5.106	error::get_ssl_category	858
5.107	error::get_system_category	858
5.108	error::make_error_code	859

5.108.1	error::make_error_code (1 of 5 overloads)	859
5.108.2	error::make_error_code (2 of 5 overloads)	859
5.108.3	error::make_error_code (3 of 5 overloads)	859
5.108.4	error::make_error_code (4 of 5 overloads)	859
5.108.5	error::make_error_code (5 of 5 overloads)	860
5.109	error::misc_category	860
5.110	error::misc_errors	860
5.111	error::netdb_category	860
5.112	error::netdb_errors	860
5.113	error::ssl_category	861
5.114	error::ssl_errors	861
5.115	error::system_category	861
5.116	error_category	861
5.116.1	error_category::message	862
5.116.2	error_category::name	862
5.116.3	error_category::operator!=	862
5.116.4	error_category::operator==	862
5.116.5	error_category::~error_category	862
5.117	error_code	863
5.117.1	error_code::category	863
5.117.2	error_code::error_code	864
5.117.2.1	error_code::error_code (1 of 3 overloads)	864
5.117.2.2	error_code::error_code (2 of 3 overloads)	864
5.117.2.3	error_code::error_code (3 of 3 overloads)	864
5.117.3	error_code::message	864
5.117.4	error_code::operator unspecified_bool_type	864
5.117.5	error_code::operator!	865
5.117.6	error_code::operator!=	865
5.117.7	error_code::operator==	865
5.117.8	error_code::unspecified_bool_true	865
5.117.9	error_code::unspecified_bool_type	865
5.117.10	error_code::value	866
5.118	error_code::unspecified_bool_type_t	866
5.119	generic::basic_endpoint	866
5.119.1	generic::basic_endpoint::basic_endpoint	867
5.119.1.1	generic::basic_endpoint::basic_endpoint (1 of 4 overloads)	868
5.119.1.2	generic::basic_endpoint::basic_endpoint (2 of 4 overloads)	868
5.119.1.3	generic::basic_endpoint::basic_endpoint (3 of 4 overloads)	868
5.119.1.4	generic::basic_endpoint::basic_endpoint (4 of 4 overloads)	868

5.119.2	generic::basic_endpoint::capacity	868
5.119.3	generic::basic_endpoint::data	869
5.119.3.1	generic::basic_endpoint::data (1 of 2 overloads)	869
5.119.3.2	generic::basic_endpoint::data (2 of 2 overloads)	869
5.119.4	generic::basic_endpoint::data_type	869
5.119.5	generic::basic_endpoint::operator!=	869
5.119.6	generic::basic_endpoint::operator<	869
5.119.7	generic::basic_endpoint::operator<=	870
5.119.8	generic::basic_endpoint::operator=	870
5.119.9	generic::basic_endpoint::operator==	870
5.119.10	generic::basic_endpoint::operator>	870
5.119.11	generic::basic_endpoint::operator>=	871
5.119.12	generic::basic_endpoint::protocol	871
5.119.13	generic::basic_endpoint::protocol_type	871
5.119.14	generic::basic_endpoint::resize	871
5.119.15	generic::basic_endpoint::size	871
5.120	generic::datagram_protocol	871
5.120.1	generic::datagram_protocol::datagram_protocol	873
5.120.1.1	generic::datagram_protocol::datagram_protocol (1 of 2 overloads)	873
5.120.1.2	generic::datagram_protocol::datagram_protocol (2 of 2 overloads)	873
5.120.2	generic::datagram_protocol::endpoint	873
5.120.3	generic::datagram_protocol::family	875
5.120.4	generic::datagram_protocol::operator!=	875
5.120.5	generic::datagram_protocol::operator==	875
5.120.6	generic::datagram_protocol::protocol	875
5.120.7	generic::datagram_protocol::socket	875
5.120.8	generic::datagram_protocol::type	879
5.121	generic::raw_protocol	879
5.121.1	generic::raw_protocol::endpoint	881
5.121.2	generic::raw_protocol::family	882
5.121.3	generic::raw_protocol::operator!=	882
5.121.4	generic::raw_protocol::operator==	882
5.121.5	generic::raw_protocol::protocol	883
5.121.6	generic::raw_protocol::raw_protocol	883
5.121.6.1	generic::raw_protocol::raw_protocol (1 of 2 overloads)	883
5.121.6.2	generic::raw_protocol::raw_protocol (2 of 2 overloads)	883
5.121.7	generic::raw_protocol::socket	883
5.121.8	generic::raw_protocol::type	887
5.122	generic::seq_packet_protocol	887

5.122.1	generic::seq_packet_protocol::endpoint	888
5.122.2	generic::seq_packet_protocol::family	890
5.122.3	generic::seq_packet_protocol::operator!=	890
5.122.4	generic::seq_packet_protocol::operator==	890
5.122.5	generic::seq_packet_protocol::protocol	890
5.122.6	generic::seq_packet_protocol::seq_packet_protocol	891
5.122.6.1	generic::seq_packet_protocol::seq_packet_protocol (1 of 2 overloads)	891
5.122.6.2	generic::seq_packet_protocol::seq_packet_protocol (2 of 2 overloads)	891
5.122.7	generic::seq_packet_protocol::socket	891
5.122.8	generic::seq_packet_protocol::type	895
5.123	generic::stream_protocol	895
5.123.1	generic::stream_protocol::endpoint	896
5.123.2	generic::stream_protocol::family	898
5.123.3	generic::stream_protocol::iostream	898
5.123.4	generic::stream_protocol::operator!=	899
5.123.5	generic::stream_protocol::operator==	899
5.123.6	generic::stream_protocol::protocol	899
5.123.7	generic::stream_protocol::socket	899
5.123.8	generic::stream_protocol::stream_protocol	903
5.123.8.1	generic::stream_protocol::stream_protocol (1 of 2 overloads)	903
5.123.8.2	generic::stream_protocol::stream_protocol (2 of 2 overloads)	904
5.123.9	generic::stream_protocol::type	904
5.124	handler_type	904
5.124.1	handler_type::type	904
5.125	has_service	905
5.126	high_resolution_timer	905
5.127	invalid_service_owner	908
5.127.1	invalid_service_owner::invalid_service_owner	908
5.128	io_service	908
5.128.1	io_service::add_service	912
5.128.2	io_service::dispatch	912
5.128.3	io_service::fork_event	913
5.128.4	io_service::has_service	913
5.128.5	io_service::io_service	914
5.128.5.1	io_service::io_service (1 of 2 overloads)	914
5.128.5.2	io_service::io_service (2 of 2 overloads)	914
5.128.6	io_service::notify_fork	914
5.128.7	io_service::poll	915
5.128.7.1	io_service::poll (1 of 2 overloads)	915

5.128.7.2	io_service::poll (2 of 2 overloads)	916
5.128.8	io_service::poll_one	916
5.128.8.1	io_service::poll_one (1 of 2 overloads)	916
5.128.8.2	io_service::poll_one (2 of 2 overloads)	916
5.128.9	io_service::post	917
5.128.10	io_service::reset	917
5.128.11	io_service::run	918
5.128.11.1	io_service::run (1 of 2 overloads)	918
5.128.11.2	io_service::run (2 of 2 overloads)	918
5.128.12	io_service::run_one	919
5.128.12.1	io_service::run_one (1 of 2 overloads)	919
5.128.12.2	io_service::run_one (2 of 2 overloads)	919
5.128.13	io_service::stop	920
5.128.14	io_service::stopped	920
5.128.15	io_service::use_service	920
5.128.16	io_service::wrap	921
5.128.17	io_service::~io_service	921
5.129	io_service::id	922
5.129.1	io_service::id::id	922
5.130	io_service::service	922
5.130.1	io_service::service::get_io_service	923
5.130.2	io_service::service::service	923
5.130.3	io_service::service::~service	924
5.130.4	io_service::service::fork_service	924
5.130.5	io_service::service::shutdown_service	924
5.131	io_service::strand	924
5.131.1	io_service::strand::dispatch	926
5.131.2	io_service::strand::get_io_service	926
5.131.3	io_service::strand::post	926
5.131.4	io_service::strand::running_in_this_thread	927
5.131.5	io_service::strand::strand	927
5.131.6	io_service::strand::wrap	927
5.131.7	io_service::strand::~strand	928
5.132	io_service::work	928
5.132.1	io_service::work::get_io_service	929
5.132.2	io_service::work::work	929
5.132.2.1	io_service::work::work (1 of 2 overloads)	929
5.132.2.2	io_service::work::work (2 of 2 overloads)	929
5.132.3	io_service::work::~work	929

5.133 ip::address	930
5.133.1 ip::address::address	931
5.133.1.1 ip::address::address (1 of 4 overloads)	931
5.133.1.2 ip::address::address (2 of 4 overloads)	932
5.133.1.3 ip::address::address (3 of 4 overloads)	932
5.133.1.4 ip::address::address (4 of 4 overloads)	932
5.133.2 ip::address::from_string	932
5.133.2.1 ip::address::from_string (1 of 4 overloads)	932
5.133.2.2 ip::address::from_string (2 of 4 overloads)	932
5.133.2.3 ip::address::from_string (3 of 4 overloads)	933
5.133.2.4 ip::address::from_string (4 of 4 overloads)	933
5.133.3 ip::address::is_loopback	933
5.133.4 ip::address::is_multicast	933
5.133.5 ip::address::is_unspecified	933
5.133.6 ip::address::is_v4	933
5.133.7 ip::address::is_v6	933
5.133.8 ip::address::operator!=	933
5.133.9 ip::address::operator<	934
5.133.10 ip::address::operator<<	934
5.133.11 ip::address::operator<=	934
5.133.12 ip::address::operator=	935
5.133.12.1 ip::address::operator= (1 of 3 overloads)	935
5.133.12.2 ip::address::operator= (2 of 3 overloads)	935
5.133.12.3 ip::address::operator= (3 of 3 overloads)	935
5.133.13 ip::address::operator==	935
5.133.14 ip::address::operator>	936
5.133.15 ip::address::operator>=	936
5.133.16 ip::address::to_string	936
5.133.16.1 ip::address::to_string (1 of 2 overloads)	936
5.133.16.2 ip::address::to_string (2 of 2 overloads)	936
5.133.17 ip::address::to_v4	937
5.133.18 ip::address::to_v6	937
5.134 ip::address_v4	937
5.134.1 ip::address_v4::address_v4	939
5.134.1.1 ip::address_v4::address_v4 (1 of 4 overloads)	939
5.134.1.2 ip::address_v4::address_v4 (2 of 4 overloads)	939
5.134.1.3 ip::address_v4::address_v4 (3 of 4 overloads)	939
5.134.1.4 ip::address_v4::address_v4 (4 of 4 overloads)	939
5.134.2 ip::address_v4::any	940

5.134.3	ip::address_v4::broadcast	940
5.134.3.1	ip::address_v4::broadcast (1 of 2 overloads)	940
5.134.3.2	ip::address_v4::broadcast (2 of 2 overloads)	940
5.134.4	ip::address_v4::bytes_type	940
5.134.5	ip::address_v4::from_string	941
5.134.5.1	ip::address_v4::from_string (1 of 4 overloads)	941
5.134.5.2	ip::address_v4::from_string (2 of 4 overloads)	941
5.134.5.3	ip::address_v4::from_string (3 of 4 overloads)	941
5.134.5.4	ip::address_v4::from_string (4 of 4 overloads)	941
5.134.6	ip::address_v4::is_class_a	941
5.134.7	ip::address_v4::is_class_b	942
5.134.8	ip::address_v4::is_class_c	942
5.134.9	ip::address_v4::is_loopback	942
5.134.10	ip::address_v4::is_multicast	942
5.134.11	ip::address_v4::is_unspecified	942
5.134.12	ip::address_v4::loopback	942
5.134.13	ip::address_v4::netmask	942
5.134.14	ip::address_v4::operator!=	942
5.134.15	ip::address_v4::operator<	943
5.134.16	ip::address_v4::operator<<	943
5.134.17	ip::address_v4::operator<=	943
5.134.18	ip::address_v4::operator=	944
5.134.19	ip::address_v4::operator==	944
5.134.20	ip::address_v4::operator>	944
5.134.21	ip::address_v4::operator>=	944
5.134.22	ip::address_v4::to_bytes	944
5.134.23	ip::address_v4::to_string	945
5.134.23.1	ip::address_v4::to_string (1 of 2 overloads)	945
5.134.23.2	ip::address_v4::to_string (2 of 2 overloads)	945
5.134.24	ip::address_v4::to_ulong	945
5.135	ip::address_v6	945
5.135.1	ip::address_v6::address_v6	947
5.135.1.1	ip::address_v6::address_v6 (1 of 3 overloads)	948
5.135.1.2	ip::address_v6::address_v6 (2 of 3 overloads)	948
5.135.1.3	ip::address_v6::address_v6 (3 of 3 overloads)	948
5.135.2	ip::address_v6::any	948
5.135.3	ip::address_v6::bytes_type	948
5.135.4	ip::address_v6::from_string	948
5.135.4.1	ip::address_v6::from_string (1 of 4 overloads)	949

5.135.4.2	ip::address_v6::from_string (2 of 4 overloads)	949
5.135.4.3	ip::address_v6::from_string (3 of 4 overloads)	949
5.135.4.4	ip::address_v6::from_string (4 of 4 overloads)	949
5.135.5	ip::address_v6::is_link_local	949
5.135.6	ip::address_v6::is_loopback	949
5.135.7	ip::address_v6::is_multicast	950
5.135.8	ip::address_v6::is_multicast_global	950
5.135.9	ip::address_v6::is_multicast_link_local	950
5.135.10	ip::address_v6::is_multicast_node_local	950
5.135.11	ip::address_v6::is_multicast_org_local	950
5.135.12	ip::address_v6::is_multicast_site_local	950
5.135.13	ip::address_v6::is_site_local	950
5.135.14	ip::address_v6::is_unspecified	950
5.135.15	ip::address_v6::is_v4_compatible	951
5.135.16	ip::address_v6::is_v4_mapped	951
5.135.17	ip::address_v6::loopback	951
5.135.18	ip::address_v6::operator!=	951
5.135.19	ip::address_v6::operator<	951
5.135.20	ip::address_v6::operator<<	951
5.135.21	ip::address_v6::operator<=	952
5.135.22	ip::address_v6::operator=	952
5.135.23	ip::address_v6::operator==	952
5.135.24	ip::address_v6::operator>	952
5.135.25	ip::address_v6::operator>=	953
5.135.26	ip::address_v6::scope_id	953
5.135.26.1	ip::address_v6::scope_id (1 of 2 overloads)	953
5.135.26.2	ip::address_v6::scope_id (2 of 2 overloads)	953
5.135.27	ip::address_v6::to_bytes	953
5.135.28	ip::address_v6::to_string	954
5.135.28.1	ip::address_v6::to_string (1 of 2 overloads)	954
5.135.28.2	ip::address_v6::to_string (2 of 2 overloads)	954
5.135.29	ip::address_v6::to_v4	954
5.135.30	ip::address_v6::v4_compatible	954
5.135.31	ip::address_v6::v4_mapped	954
5.136	ip::basic_endpoint	954
5.136.1	ip::basic_endpoint::address	956
5.136.1.1	ip::basic_endpoint::address (1 of 2 overloads)	956
5.136.1.2	ip::basic_endpoint::address (2 of 2 overloads)	956
5.136.2	ip::basic_endpoint::basic_endpoint	957

5.136.2.1	ip::basic_endpoint::basic_endpoint (1 of 4 overloads)	957
5.136.2.2	ip::basic_endpoint::basic_endpoint (2 of 4 overloads)	957
5.136.2.3	ip::basic_endpoint::basic_endpoint (3 of 4 overloads)	958
5.136.2.4	ip::basic_endpoint::basic_endpoint (4 of 4 overloads)	958
5.136.3	ip::basic_endpoint::capacity	958
5.136.4	ip::basic_endpoint::data	958
5.136.4.1	ip::basic_endpoint::data (1 of 2 overloads)	958
5.136.4.2	ip::basic_endpoint::data (2 of 2 overloads)	958
5.136.5	ip::basic_endpoint::data_type	958
5.136.6	ip::basic_endpoint::operator!=	959
5.136.7	ip::basic_endpoint::operator<	959
5.136.8	ip::basic_endpoint::operator<<	959
5.136.9	ip::basic_endpoint::operator<=	959
5.136.10	ip::basic_endpoint::operator=	960
5.136.11	ip::basic_endpoint::operator==	960
5.136.12	ip::basic_endpoint::operator>	960
5.136.13	ip::basic_endpoint::operator>=	960
5.136.14	ip::basic_endpoint::port	961
5.136.14.1	ip::basic_endpoint::port (1 of 2 overloads)	961
5.136.14.2	ip::basic_endpoint::port (2 of 2 overloads)	961
5.136.15	ip::basic_endpoint::protocol	961
5.136.16	ip::basic_endpoint::protocol_type	961
5.136.17	ip::basic_endpoint::resize	961
5.136.18	ip::basic_endpoint::size	962
5.137	ip::basic_resolver	962
5.137.1	ip::basic_resolver::async_resolve	963
5.137.1.1	ip::basic_resolver::async_resolve (1 of 2 overloads)	964
5.137.1.2	ip::basic_resolver::async_resolve (2 of 2 overloads)	964
5.137.2	ip::basic_resolver::basic_resolver	965
5.137.3	ip::basic_resolver::cancel	965
5.137.4	ip::basic_resolver::endpoint_type	965
5.137.5	ip::basic_resolver::get_implementation	965
5.137.5.1	ip::basic_resolver::get_implementation (1 of 2 overloads)	966
5.137.5.2	ip::basic_resolver::get_implementation (2 of 2 overloads)	966
5.137.6	ip::basic_resolver::get_io_service	966
5.137.7	ip::basic_resolver::get_service	966
5.137.7.1	ip::basic_resolver::get_service (1 of 2 overloads)	966
5.137.7.2	ip::basic_resolver::get_service (2 of 2 overloads)	966
5.137.8	ip::basic_resolver::implementation	967

5.137.9	ip::basic_resolver::implementation_type	967
5.137.10	ip::basic_resolver::iterator	967
5.137.11	ip::basic_resolver::protocol_type	968
5.137.12	ip::basic_resolver::query	969
5.137.13	ip::basic_resolver::resolve	970
5.137.13.1	ip::basic_resolver::resolve (1 of 4 overloads)	970
5.137.13.2	ip::basic_resolver::resolve (2 of 4 overloads)	971
5.137.13.3	ip::basic_resolver::resolve (3 of 4 overloads)	971
5.137.13.4	ip::basic_resolver::resolve (4 of 4 overloads)	972
5.137.14	ip::basic_resolver::service	972
5.137.15	ip::basic_resolver::service_type	973
5.138	ip::basic_resolver_entry	973
5.138.1	ip::basic_resolver_entry::basic_resolver_entry	974
5.138.1.1	ip::basic_resolver_entry::basic_resolver_entry (1 of 2 overloads)	974
5.138.1.2	ip::basic_resolver_entry::basic_resolver_entry (2 of 2 overloads)	974
5.138.2	ip::basic_resolver_entry::endpoint	974
5.138.3	ip::basic_resolver_entry::endpoint_type	974
5.138.4	ip::basic_resolver_entry::host_name	975
5.138.5	ip::basic_resolver_entry::operator endpoint_type	975
5.138.6	ip::basic_resolver_entry::protocol_type	975
5.138.7	ip::basic_resolver_entry::service_name	975
5.139	ip::basic_resolver_iterator	975
5.139.1	ip::basic_resolver_iterator::basic_resolver_iterator	977
5.139.2	ip::basic_resolver_iterator::create	977
5.139.2.1	ip::basic_resolver_iterator::create (1 of 3 overloads)	977
5.139.2.2	ip::basic_resolver_iterator::create (2 of 3 overloads)	978
5.139.2.3	ip::basic_resolver_iterator::create (3 of 3 overloads)	978
5.139.3	ip::basic_resolver_iterator::difference_type	978
5.139.4	ip::basic_resolver_iterator::iterator_category	978
5.139.5	ip::basic_resolver_iterator::operator *	978
5.139.6	ip::basic_resolver_iterator::operator!=	979
5.139.7	ip::basic_resolver_iterator::operator++	979
5.139.7.1	ip::basic_resolver_iterator::operator++ (1 of 2 overloads)	979
5.139.7.2	ip::basic_resolver_iterator::operator++ (2 of 2 overloads)	979
5.139.8	ip::basic_resolver_iterator::operator->	979
5.139.9	ip::basic_resolver_iterator::operator==	979
5.139.10	ip::basic_resolver_iterator::pointer	980
5.139.11	ip::basic_resolver_iterator::reference	980
5.139.12	ip::basic_resolver_iterator::value_type	981

5.140	ip::basic_resolver_query	982
5.140.1	ip::basic_resolver_query::address_configured	983
5.140.2	ip::basic_resolver_query::all_matching	983
5.140.3	ip::basic_resolver_query::basic_resolver_query	984
5.140.3.1	ip::basic_resolver_query::basic_resolver_query (1 of 4 overloads)	984
5.140.3.2	ip::basic_resolver_query::basic_resolver_query (2 of 4 overloads)	985
5.140.3.3	ip::basic_resolver_query::basic_resolver_query (3 of 4 overloads)	985
5.140.3.4	ip::basic_resolver_query::basic_resolver_query (4 of 4 overloads)	986
5.140.4	ip::basic_resolver_query::canonical_name	986
5.140.5	ip::basic_resolver_query::flags	986
5.140.6	ip::basic_resolver_query::hints	987
5.140.7	ip::basic_resolver_query::host_name	987
5.140.8	ip::basic_resolver_query::numeric_host	987
5.140.9	ip::basic_resolver_query::numeric_service	987
5.140.10	ip::basic_resolver_query::passive	987
5.140.11	ip::basic_resolver_query::protocol_type	987
5.140.12	ip::basic_resolver_query::service_name	987
5.140.13	ip::basic_resolver_query::v4_mapped	988
5.141	ip::host_name	988
5.141.1	ip::host_name (1 of 2 overloads)	988
5.141.2	ip::host_name (2 of 2 overloads)	988
5.142	ip::icmp	988
5.142.1	ip::icmp::endpoint	989
5.142.2	ip::icmp::family	991
5.142.3	ip::icmp::operator!=	991
5.142.4	ip::icmp::operator==	991
5.142.5	ip::icmp::protocol	992
5.142.6	ip::icmp::resolver	992
5.142.7	ip::icmp::socket	993
5.142.8	ip::icmp::type	997
5.142.9	ip::icmp::v4	997
5.142.10	ip::icmp::v6	997
5.143	ip::multicast::enable_loopback	997
5.144	ip::multicast::hops	998
5.145	ip::multicast::join_group	999
5.146	ip::multicast::leave_group	999
5.147	ip::multicast::outbound_interface	999
5.148	ip::resolver_query_base	1000
5.148.1	ip::resolver_query_base::address_configured	1001

5.148.2	ip::resolver_query_base::all_matching	1001
5.148.3	ip::resolver_query_base::canonical_name	1001
5.148.4	ip::resolver_query_base::flags	1001
5.148.5	ip::resolver_query_base::numeric_host	1002
5.148.6	ip::resolver_query_base::numeric_service	1002
5.148.7	ip::resolver_query_base::passive	1002
5.148.8	ip::resolver_query_base::v4_mapped	1002
5.148.9	ip::resolver_query_base::~resolver_query_base	1002
5.149	ip::resolver_service	1002
5.149.1	ip::resolver_service::async_resolve	1003
5.149.1.1	ip::resolver_service::async_resolve (1 of 2 overloads)	1004
5.149.1.2	ip::resolver_service::async_resolve (2 of 2 overloads)	1004
5.149.2	ip::resolver_service::cancel	1004
5.149.3	ip::resolver_service::construct	1004
5.149.4	ip::resolver_service::destroy	1004
5.149.5	ip::resolver_service::endpoint_type	1004
5.149.6	ip::resolver_service::get_io_service	1005
5.149.7	ip::resolver_service::id	1005
5.149.8	ip::resolver_service::implementation_type	1005
5.149.9	ip::resolver_service::iterator_type	1005
5.149.10	ip::resolver_service::protocol_type	1006
5.149.11	ip::resolver_service::query_type	1007
5.149.12	ip::resolver_service::resolve	1008
5.149.12.1	ip::resolver_service::resolve (1 of 2 overloads)	1008
5.149.12.2	ip::resolver_service::resolve (2 of 2 overloads)	1009
5.149.13	ip::resolver_service::resolver_service	1009
5.150	ip::tcp	1009
5.150.1	ip::tcp::acceptor	1010
5.150.2	ip::tcp::endpoint	1013
5.150.3	ip::tcp::family	1015
5.150.4	ip::tcp::iostream	1015
5.150.5	ip::tcp::no_delay	1016
5.150.6	ip::tcp::operator!=	1017
5.150.7	ip::tcp::operator==	1017
5.150.8	ip::tcp::protocol	1017
5.150.9	ip::tcp::resolver	1017
5.150.10	ip::tcp::socket	1019
5.150.11	ip::tcp::type	1023
5.150.12	ip::tcp::v4	1023

5.150.13	ip::tcp::v6	1023
5.151	ip::udp	1023
5.151.1	ip::udp::endpoint	1024
5.151.2	ip::udp::family	1026
5.151.3	ip::udp::operator!=	1026
5.151.4	ip::udp::operator==	1026
5.151.5	ip::udp::protocol	1027
5.151.6	ip::udp::resolver	1027
5.151.7	ip::udp::socket	1028
5.151.8	ip::udp::type	1032
5.151.9	ip::udp::v4	1032
5.151.10	ip::udp::v6	1032
5.152	ip::unicast::hops	1032
5.153	ip::v6_only	1033
5.154	is_match_condition	1034
5.154.1	is_match_condition::value	1034
5.155	is_read_buffered	1034
5.155.1	is_read_buffered::value	1035
5.156	is_write_buffered	1035
5.156.1	is_write_buffered::value	1035
5.157	local::basic_endpoint	1035
5.157.1	local::basic_endpoint::basic_endpoint	1037
5.157.1.1	local::basic_endpoint::basic_endpoint (1 of 4 overloads)	1037
5.157.1.2	local::basic_endpoint::basic_endpoint (2 of 4 overloads)	1037
5.157.1.3	local::basic_endpoint::basic_endpoint (3 of 4 overloads)	1037
5.157.1.4	local::basic_endpoint::basic_endpoint (4 of 4 overloads)	1037
5.157.2	local::basic_endpoint::capacity	1038
5.157.3	local::basic_endpoint::data	1038
5.157.3.1	local::basic_endpoint::data (1 of 2 overloads)	1038
5.157.3.2	local::basic_endpoint::data (2 of 2 overloads)	1038
5.157.4	local::basic_endpoint::data_type	1038
5.157.5	local::basic_endpoint::operator!=	1038
5.157.6	local::basic_endpoint::operator<	1039
5.157.7	local::basic_endpoint::operator<<	1039
5.157.8	local::basic_endpoint::operator<=	1039
5.157.9	local::basic_endpoint::operator=	1039
5.157.10	local::basic_endpoint::operator==	1040
5.157.11	local::basic_endpoint::operator>	1040
5.157.12	local::basic_endpoint::operator>=	1040

5.157.13	local::basic_endpoint::path	1040
5.157.13.1	local::basic_endpoint::path (1 of 3 overloads)	1041
5.157.13.2	local::basic_endpoint::path (2 of 3 overloads)	1041
5.157.13.3	local::basic_endpoint::path (3 of 3 overloads)	1041
5.157.14	local::basic_endpoint::protocol	1041
5.157.15	local::basic_endpoint::protocol_type	1041
5.157.16	local::basic_endpoint::resize	1041
5.157.17	local::basic_endpoint::size	1041
5.158	local::connect_pair	1042
5.158.1	local::connect_pair (1 of 2 overloads)	1042
5.158.2	local::connect_pair (2 of 2 overloads)	1042
5.159	local::datagram_protocol	1043
5.159.1	local::datagram_protocol::endpoint	1043
5.159.2	local::datagram_protocol::family	1045
5.159.3	local::datagram_protocol::protocol	1045
5.159.4	local::datagram_protocol::socket	1045
5.159.5	local::datagram_protocol::type	1049
5.160	local::stream_protocol	1049
5.160.1	local::stream_protocol::acceptor	1050
5.160.2	local::stream_protocol::endpoint	1053
5.160.3	local::stream_protocol::family	1055
5.160.4	local::stream_protocol::iostream	1055
5.160.5	local::stream_protocol::protocol	1056
5.160.6	local::stream_protocol::socket	1056
5.160.7	local::stream_protocol::type	1060
5.161	mutable_buffer	1060
5.161.1	mutable_buffer::mutable_buffer	1061
5.161.1.1	mutable_buffer::mutable_buffer (1 of 2 overloads)	1061
5.161.1.2	mutable_buffer::mutable_buffer (2 of 2 overloads)	1061
5.161.2	mutable_buffer::operator+	1062
5.161.2.1	mutable_buffer::operator+ (1 of 2 overloads)	1062
5.161.2.2	mutable_buffer::operator+ (2 of 2 overloads)	1062
5.162	mutable_buffers_1	1062
5.162.1	mutable_buffers_1::begin	1063
5.162.2	mutable_buffers_1::const_iterator	1063
5.162.3	mutable_buffers_1::end	1063
5.162.4	mutable_buffers_1::mutable_buffers_1	1064
5.162.4.1	mutable_buffers_1::mutable_buffers_1 (1 of 2 overloads)	1064
5.162.4.2	mutable_buffers_1::mutable_buffers_1 (2 of 2 overloads)	1064

5.162.5	mutable_buffers_1::operator+	1064
5.162.5.1	mutable_buffers_1::operator+ (1 of 2 overloads)	1064
5.162.5.2	mutable_buffers_1::operator+ (2 of 2 overloads)	1065
5.162.6	mutable_buffers_1::value_type	1065
5.163	null_buffers	1066
5.163.1	null_buffers::begin	1066
5.163.2	null_buffers::const_iterator	1066
5.163.3	null_buffers::end	1067
5.163.4	null_buffers::value_type	1067
5.164	operator<<	1067
5.165	placeholders::bytes_transferred	1068
5.166	placeholders::error	1068
5.167	placeholders::iterator	1068
5.168	placeholders::signal_number	1069
5.169	posix::basic_descriptor	1069
5.169.1	posix::basic_descriptor::assign	1071
5.169.1.1	posix::basic_descriptor::assign (1 of 2 overloads)	1071
5.169.1.2	posix::basic_descriptor::assign (2 of 2 overloads)	1071
5.169.2	posix::basic_descriptor::basic_descriptor	1072
5.169.2.1	posix::basic_descriptor::basic_descriptor (1 of 3 overloads)	1072
5.169.2.2	posix::basic_descriptor::basic_descriptor (2 of 3 overloads)	1072
5.169.2.3	posix::basic_descriptor::basic_descriptor (3 of 3 overloads)	1073
5.169.3	posix::basic_descriptor::bytes_readable	1073
5.169.4	posix::basic_descriptor::cancel	1073
5.169.4.1	posix::basic_descriptor::cancel (1 of 2 overloads)	1074
5.169.4.2	posix::basic_descriptor::cancel (2 of 2 overloads)	1074
5.169.5	posix::basic_descriptor::close	1074
5.169.5.1	posix::basic_descriptor::close (1 of 2 overloads)	1074
5.169.5.2	posix::basic_descriptor::close (2 of 2 overloads)	1075
5.169.6	posix::basic_descriptor::get_implementation	1075
5.169.6.1	posix::basic_descriptor::get_implementation (1 of 2 overloads)	1075
5.169.6.2	posix::basic_descriptor::get_implementation (2 of 2 overloads)	1075
5.169.7	posix::basic_descriptor::get_io_service	1075
5.169.8	posix::basic_descriptor::get_service	1076
5.169.8.1	posix::basic_descriptor::get_service (1 of 2 overloads)	1076
5.169.8.2	posix::basic_descriptor::get_service (2 of 2 overloads)	1076
5.169.9	posix::basic_descriptor::implementation	1076
5.169.10	posix::basic_descriptor::implementation_type	1076
5.169.11	posix::basic_descriptor::io_control	1077

5.169.11.1	posix::basic_descriptor::io_control (1 of 2 overloads)	1077
5.169.11.2	posix::basic_descriptor::io_control (2 of 2 overloads)	1077
5.169.12	posix::basic_descriptor::is_open	1078
5.169.13	posix::basic_descriptor::lowest_layer	1078
5.169.13.1	posix::basic_descriptor::lowest_layer (1 of 2 overloads)	1078
5.169.13.2	posix::basic_descriptor::lowest_layer (2 of 2 overloads)	1079
5.169.14	posix::basic_descriptor::lowest_layer_type	1079
5.169.15	posix::basic_descriptor::native	1081
5.169.16	posix::basic_descriptor::native_handle	1081
5.169.17	posix::basic_descriptor::native_handle_type	1081
5.169.18	posix::basic_descriptor::native_non_blocking	1081
5.169.18.1	posix::basic_descriptor::native_non_blocking (1 of 3 overloads)	1082
5.169.18.2	posix::basic_descriptor::native_non_blocking (2 of 3 overloads)	1082
5.169.18.3	posix::basic_descriptor::native_non_blocking (3 of 3 overloads)	1082
5.169.19	posix::basic_descriptor::native_type	1083
5.169.20	posix::basic_descriptor::non_blocking	1083
5.169.20.1	posix::basic_descriptor::non_blocking (1 of 3 overloads)	1083
5.169.20.2	posix::basic_descriptor::non_blocking (2 of 3 overloads)	1084
5.169.20.3	posix::basic_descriptor::non_blocking (3 of 3 overloads)	1084
5.169.21	posix::basic_descriptor::non_blocking_io	1084
5.169.22	posix::basic_descriptor::operator=	1085
5.169.23	posix::basic_descriptor::release	1085
5.169.24	posix::basic_descriptor::service	1085
5.169.25	posix::basic_descriptor::service_type	1086
5.169.26	posix::basic_descriptor::~basic_descriptor	1086
5.170	posix::basic_stream_descriptor	1086
5.170.1	posix::basic_stream_descriptor::assign	1088
5.170.1.1	posix::basic_stream_descriptor::assign (1 of 2 overloads)	1088
5.170.1.2	posix::basic_stream_descriptor::assign (2 of 2 overloads)	1089
5.170.2	posix::basic_stream_descriptor::async_read_some	1089
5.170.3	posix::basic_stream_descriptor::async_write_some	1090
5.170.4	posix::basic_stream_descriptor::basic_stream_descriptor	1090
5.170.4.1	posix::basic_stream_descriptor::basic_stream_descriptor (1 of 3 overloads)	1091
5.170.4.2	posix::basic_stream_descriptor::basic_stream_descriptor (2 of 3 overloads)	1091
5.170.4.3	posix::basic_stream_descriptor::basic_stream_descriptor (3 of 3 overloads)	1091
5.170.5	posix::basic_stream_descriptor::bytes_readable	1092
5.170.6	posix::basic_stream_descriptor::cancel	1092
5.170.6.1	posix::basic_stream_descriptor::cancel (1 of 2 overloads)	1092
5.170.6.2	posix::basic_stream_descriptor::cancel (2 of 2 overloads)	1093

5.170.7	<code>posix::basic_stream_descriptor::close</code>	1093
5.170.7.1	<code>posix::basic_stream_descriptor::close</code> (1 of 2 overloads)	1093
5.170.7.2	<code>posix::basic_stream_descriptor::close</code> (2 of 2 overloads)	1093
5.170.8	<code>posix::basic_stream_descriptor::get_implementation</code>	1094
5.170.8.1	<code>posix::basic_stream_descriptor::get_implementation</code> (1 of 2 overloads)	1094
5.170.8.2	<code>posix::basic_stream_descriptor::get_implementation</code> (2 of 2 overloads)	1094
5.170.9	<code>posix::basic_stream_descriptor::get_io_service</code>	1094
5.170.10	<code>posix::basic_stream_descriptor::get_service</code>	1094
5.170.10.1	<code>posix::basic_stream_descriptor::get_service</code> (1 of 2 overloads)	1094
5.170.10.2	<code>posix::basic_stream_descriptor::get_service</code> (2 of 2 overloads)	1095
5.170.11	<code>posix::basic_stream_descriptor::implementation</code>	1095
5.170.12	<code>posix::basic_stream_descriptor::implementation_type</code>	1095
5.170.13	<code>posix::basic_stream_descriptor::io_control</code>	1095
5.170.13.1	<code>posix::basic_stream_descriptor::io_control</code> (1 of 2 overloads)	1095
5.170.13.2	<code>posix::basic_stream_descriptor::io_control</code> (2 of 2 overloads)	1096
5.170.14	<code>posix::basic_stream_descriptor::is_open</code>	1097
5.170.15	<code>posix::basic_stream_descriptor::lowest_layer</code>	1097
5.170.15.1	<code>posix::basic_stream_descriptor::lowest_layer</code> (1 of 2 overloads)	1097
5.170.15.2	<code>posix::basic_stream_descriptor::lowest_layer</code> (2 of 2 overloads)	1097
5.170.16	<code>posix::basic_stream_descriptor::lowest_layer_type</code>	1097
5.170.17	<code>posix::basic_stream_descriptor::native</code>	1099
5.170.18	<code>posix::basic_stream_descriptor::native_handle</code>	1100
5.170.19	<code>posix::basic_stream_descriptor::native_handle_type</code>	1100
5.170.20	<code>posix::basic_stream_descriptor::native_non_blocking</code>	1100
5.170.20.1	<code>posix::basic_stream_descriptor::native_non_blocking</code> (1 of 3 overloads)	1100
5.170.20.2	<code>posix::basic_stream_descriptor::native_non_blocking</code> (2 of 3 overloads)	1101
5.170.20.3	<code>posix::basic_stream_descriptor::native_non_blocking</code> (3 of 3 overloads)	1101
5.170.21	<code>posix::basic_stream_descriptor::native_type</code>	1101
5.170.22	<code>posix::basic_stream_descriptor::non_blocking</code>	1102
5.170.22.1	<code>posix::basic_stream_descriptor::non_blocking</code> (1 of 3 overloads)	1102
5.170.22.2	<code>posix::basic_stream_descriptor::non_blocking</code> (2 of 3 overloads)	1102
5.170.22.3	<code>posix::basic_stream_descriptor::non_blocking</code> (3 of 3 overloads)	1103
5.170.23	<code>posix::basic_stream_descriptor::non_blocking_io</code>	1103
5.170.24	<code>posix::basic_stream_descriptor::operator=</code>	1104
5.170.25	<code>posix::basic_stream_descriptor::read_some</code>	1104
5.170.25.1	<code>posix::basic_stream_descriptor::read_some</code> (1 of 2 overloads)	1104
5.170.25.2	<code>posix::basic_stream_descriptor::read_some</code> (2 of 2 overloads)	1105
5.170.26	<code>posix::basic_stream_descriptor::release</code>	1105
5.170.27	<code>posix::basic_stream_descriptor::service</code>	1106

5.170.28	posix::basic_stream_descriptor::service_type	1106
5.170.29	posix::basic_stream_descriptor::write_some	1106
5.170.29.1	posix::basic_stream_descriptor::write_some (1 of 2 overloads)	1106
5.170.29.2	posix::basic_stream_descriptor::write_some (2 of 2 overloads)	1107
5.171	posix::descriptor_base	1108
5.171.1	posix::descriptor_base::bytes_readable	1108
5.171.2	posix::descriptor_base::non_blocking_io	1109
5.171.3	posix::descriptor_base::~descriptor_base	1109
5.172	posix::stream_descriptor	1109
5.173	posix::stream_descriptor_service	1111
5.173.1	posix::stream_descriptor_service::assign	1113
5.173.2	posix::stream_descriptor_service::async_read_some	1113
5.173.3	posix::stream_descriptor_service::async_write_some	1113
5.173.4	posix::stream_descriptor_service::cancel	1113
5.173.5	posix::stream_descriptor_service::close	1114
5.173.6	posix::stream_descriptor_service::construct	1114
5.173.7	posix::stream_descriptor_service::destroy	1114
5.173.8	posix::stream_descriptor_service::get_io_service	1114
5.173.9	posix::stream_descriptor_service::id	1114
5.173.10	posix::stream_descriptor_service::implementation_type	1114
5.173.11	posix::stream_descriptor_service::io_control	1115
5.173.12	posix::stream_descriptor_service::is_open	1115
5.173.13	posix::stream_descriptor_service::move_assign	1115
5.173.14	posix::stream_descriptor_service::move_construct	1115
5.173.15	posix::stream_descriptor_service::native	1115
5.173.16	posix::stream_descriptor_service::native_handle	1115
5.173.17	posix::stream_descriptor_service::native_handle_type	1115
5.173.18	posix::stream_descriptor_service::native_non_blocking	1116
5.173.18.1	posix::stream_descriptor_service::native_non_blocking (1 of 2 overloads)	1116
5.173.18.2	posix::stream_descriptor_service::native_non_blocking (2 of 2 overloads)	1116
5.173.19	posix::stream_descriptor_service::native_type	1116
5.173.20	posix::stream_descriptor_service::non_blocking	1117
5.173.20.1	posix::stream_descriptor_service::non_blocking (1 of 2 overloads)	1117
5.173.20.2	posix::stream_descriptor_service::non_blocking (2 of 2 overloads)	1117
5.173.21	posix::stream_descriptor_service::read_some	1117
5.173.22	posix::stream_descriptor_service::release	1117
5.173.23	posix::stream_descriptor_service::stream_descriptor_service	1117
5.173.24	posix::stream_descriptor_service::write_some	1118
5.174	raw_socket_service	1118

5.174.1	raw_socket_service::assign	1120
5.174.2	raw_socket_service::async_connect	1120
5.174.3	raw_socket_service::async_receive	1120
5.174.4	raw_socket_service::async_receive_from	1121
5.174.5	raw_socket_service::async_send	1121
5.174.6	raw_socket_service::async_send_to	1121
5.174.7	raw_socket_service::at_mark	1121
5.174.8	raw_socket_service::available	1122
5.174.9	raw_socket_service::bind	1122
5.174.10	raw_socket_service::cancel	1122
5.174.11	raw_socket_service::close	1122
5.174.12	raw_socket_service::connect	1122
5.174.13	raw_socket_service::construct	1122
5.174.14	raw_socket_service::converting_move_construct	1123
5.174.15	raw_socket_service::destroy	1123
5.174.16	raw_socket_service::endpoint_type	1123
5.174.17	raw_socket_service::get_io_service	1123
5.174.18	raw_socket_service::get_option	1123
5.174.19	raw_socket_service::id	1123
5.174.20	raw_socket_service::implementation_type	1124
5.174.21	raw_socket_service::io_control	1124
5.174.22	raw_socket_service::is_open	1124
5.174.23	raw_socket_service::local_endpoint	1124
5.174.24	raw_socket_service::move_assign	1124
5.174.25	raw_socket_service::move_construct	1124
5.174.26	raw_socket_service::native	1125
5.174.27	raw_socket_service::native_handle	1125
5.174.28	raw_socket_service::native_handle_type	1125
5.174.29	raw_socket_service::native_non_blocking	1125
5.174.29.1	raw_socket_service::native_non_blocking (1 of 2 overloads)	1125
5.174.29.2	raw_socket_service::native_non_blocking (2 of 2 overloads)	1125
5.174.30	raw_socket_service::native_type	1126
5.174.31	raw_socket_service::non_blocking	1126
5.174.31.1	raw_socket_service::non_blocking (1 of 2 overloads)	1126
5.174.31.2	raw_socket_service::non_blocking (2 of 2 overloads)	1126
5.174.32	raw_socket_service::open	1126
5.174.33	raw_socket_service::protocol_type	1126
5.174.34	raw_socket_service::raw_socket_service	1127
5.174.35	raw_socket_service::receive	1127

5.174.36	raw_socket_service::receive_from	1127
5.174.37	raw_socket_service::remote_endpoint	1127
5.174.38	raw_socket_service::send	1127
5.174.39	raw_socket_service::send_to	1128
5.174.40	raw_socket_service::set_option	1128
5.174.41	raw_socket_service::shutdown	1128
5.175	read	1128
5.175.1	read (1 of 8 overloads)	1130
5.175.2	read (2 of 8 overloads)	1131
5.175.3	read (3 of 8 overloads)	1132
5.175.4	read (4 of 8 overloads)	1133
5.175.5	read (5 of 8 overloads)	1133
5.175.6	read (6 of 8 overloads)	1134
5.175.7	read (7 of 8 overloads)	1135
5.175.8	read (8 of 8 overloads)	1136
5.176	read_at	1137
5.176.1	read_at (1 of 8 overloads)	1138
5.176.2	read_at (2 of 8 overloads)	1139
5.176.3	read_at (3 of 8 overloads)	1140
5.176.4	read_at (4 of 8 overloads)	1141
5.176.5	read_at (5 of 8 overloads)	1142
5.176.6	read_at (6 of 8 overloads)	1143
5.176.7	read_at (7 of 8 overloads)	1144
5.176.8	read_at (8 of 8 overloads)	1145
5.177	read_until	1146
5.177.1	read_until (1 of 8 overloads)	1147
5.177.2	read_until (2 of 8 overloads)	1148
5.177.3	read_until (3 of 8 overloads)	1149
5.177.4	read_until (4 of 8 overloads)	1150
5.177.5	read_until (5 of 8 overloads)	1151
5.177.6	read_until (6 of 8 overloads)	1152
5.177.7	read_until (7 of 8 overloads)	1153
5.177.8	read_until (8 of 8 overloads)	1155
5.178	seq_packet_socket_service	1156
5.178.1	seq_packet_socket_service::assign	1158
5.178.2	seq_packet_socket_service::async_connect	1158
5.178.3	seq_packet_socket_service::async_receive	1158
5.178.4	seq_packet_socket_service::async_send	1159
5.178.5	seq_packet_socket_service::at_mark	1159

5.178.6	seq_packet_socket_service::available	1159
5.178.7	seq_packet_socket_service::bind	1159
5.178.8	seq_packet_socket_service::cancel	1159
5.178.9	seq_packet_socket_service::close	1160
5.178.10	seq_packet_socket_service::connect	1160
5.178.11	seq_packet_socket_service::construct	1160
5.178.12	seq_packet_socket_service::converting_move_construct	1160
5.178.13	seq_packet_socket_service::destroy	1160
5.178.14	seq_packet_socket_service::endpoint_type	1160
5.178.15	seq_packet_socket_service::get_io_service	1161
5.178.16	seq_packet_socket_service::get_option	1161
5.178.17	seq_packet_socket_service::id	1161
5.178.18	seq_packet_socket_service::implementation_type	1161
5.178.19	seq_packet_socket_service::io_control	1161
5.178.20	seq_packet_socket_service::is_open	1161
5.178.21	seq_packet_socket_service::local_endpoint	1162
5.178.22	seq_packet_socket_service::move_assign	1162
5.178.23	seq_packet_socket_service::move_construct	1162
5.178.24	seq_packet_socket_service::native	1162
5.178.25	seq_packet_socket_service::native_handle	1162
5.178.26	seq_packet_socket_service::native_handle_type	1162
5.178.27	seq_packet_socket_service::native_non_blocking	1163
5.178.27.1	seq_packet_socket_service::native_non_blocking (1 of 2 overloads)	1163
5.178.27.2	seq_packet_socket_service::native_non_blocking (2 of 2 overloads)	1163
5.178.28	seq_packet_socket_service::native_type	1163
5.178.29	seq_packet_socket_service::non_blocking	1163
5.178.29.1	seq_packet_socket_service::non_blocking (1 of 2 overloads)	1164
5.178.29.2	seq_packet_socket_service::non_blocking (2 of 2 overloads)	1164
5.178.30	seq_packet_socket_service::open	1164
5.178.31	seq_packet_socket_service::protocol_type	1164
5.178.32	seq_packet_socket_service::receive	1164
5.178.33	seq_packet_socket_service::remote_endpoint	1164
5.178.34	seq_packet_socket_service::send	1165
5.178.35	seq_packet_socket_service::seq_packet_socket_service	1165
5.178.36	seq_packet_socket_service::set_option	1165
5.178.37	seq_packet_socket_service::shutdown	1165
5.179	serial_port	1165
5.180	serial_port_base	1167
5.180.1	serial_port_base::~serial_port_base	1168

5.181	serial_port_base::baud_rate	1168
5.181.1	serial_port_base::baud_rate::baud_rate	1169
5.181.2	serial_port_base::baud_rate::load	1169
5.181.3	serial_port_base::baud_rate::store	1169
5.181.4	serial_port_base::baud_rate::value	1169
5.182	serial_port_base::character_size	1169
5.182.1	serial_port_base::character_size::character_size	1169
5.182.2	serial_port_base::character_size::load	1170
5.182.3	serial_port_base::character_size::store	1170
5.182.4	serial_port_base::character_size::value	1170
5.183	serial_port_base::flow_control	1170
5.183.1	serial_port_base::flow_control::flow_control	1170
5.183.2	serial_port_base::flow_control::load	1171
5.183.3	serial_port_base::flow_control::store	1171
5.183.4	serial_port_base::flow_control::type	1171
5.183.5	serial_port_base::flow_control::value	1171
5.184	serial_port_base::parity	1171
5.184.1	serial_port_base::parity::load	1172
5.184.2	serial_port_base::parity::parity	1172
5.184.3	serial_port_base::parity::store	1172
5.184.4	serial_port_base::parity::type	1172
5.184.5	serial_port_base::parity::value	1172
5.185	serial_port_base::stop_bits	1172
5.185.1	serial_port_base::stop_bits::load	1173
5.185.2	serial_port_base::stop_bits::stop_bits	1173
5.185.3	serial_port_base::stop_bits::store	1173
5.185.4	serial_port_base::stop_bits::type	1173
5.185.5	serial_port_base::stop_bits::value	1174
5.186	serial_port_service	1174
5.186.1	serial_port_service::assign	1175
5.186.2	serial_port_service::async_read_some	1175
5.186.3	serial_port_service::async_write_some	1176
5.186.4	serial_port_service::cancel	1176
5.186.5	serial_port_service::close	1176
5.186.6	serial_port_service::construct	1176
5.186.7	serial_port_service::destroy	1176
5.186.8	serial_port_service::get_io_service	1176
5.186.9	serial_port_service::get_option	1177
5.186.10	serial_port_service::id	1177

5.186.11	serial_port_service::implementation_type	1177
5.186.12	serial_port_service::is_open	1177
5.186.13	serial_port_service::move_assign	1177
5.186.14	serial_port_service::move_construct	1177
5.186.15	serial_port_service::native	1178
5.186.16	serial_port_service::native_handle	1178
5.186.17	serial_port_service::native_handle_type	1178
5.186.18	serial_port_service::native_type	1178
5.186.19	serial_port_service::open	1178
5.186.20	serial_port_service::read_some	1178
5.186.21	serial_port_service::send_break	1179
5.186.22	serial_port_service::serial_port_service	1179
5.186.23	serial_port_service::set_option	1179
5.186.24	serial_port_service::write_some	1179
5.187	service_already_exists	1179
5.187.1	service_already_exists::service_already_exists	1180
5.188	signal_set	1180
5.189	signal_set_service	1182
5.189.1	signal_set_service::add	1183
5.189.2	signal_set_service::async_wait	1183
5.189.3	signal_set_service::cancel	1183
5.189.4	signal_set_service::clear	1183
5.189.5	signal_set_service::construct	1184
5.189.6	signal_set_service::destroy	1184
5.189.7	signal_set_service::get_io_service	1184
5.189.8	signal_set_service::id	1184
5.189.9	signal_set_service::implementation_type	1184
5.189.10	signal_set_service::remove	1184
5.189.11	signal_set_service::signal_set_service	1185
5.190	socket_acceptor_service	1185
5.190.1	socket_acceptor_service::accept	1187
5.190.2	socket_acceptor_service::assign	1187
5.190.3	socket_acceptor_service::async_accept	1187
5.190.4	socket_acceptor_service::bind	1187
5.190.5	socket_acceptor_service::cancel	1187
5.190.6	socket_acceptor_service::close	1188
5.190.7	socket_acceptor_service::construct	1188
5.190.8	socket_acceptor_service::converting_move_construct	1188
5.190.9	socket_acceptor_service::destroy	1188

5.190.10	socket_acceptor_service::endpoint_type	1188
5.190.11	socket_acceptor_service::get_io_service	1188
5.190.12	socket_acceptor_service::get_option	1189
5.190.13	socket_acceptor_service::id	1189
5.190.14	socket_acceptor_service::implementation_type	1189
5.190.15	socket_acceptor_service::io_control	1189
5.190.16	socket_acceptor_service::is_open	1189
5.190.17	socket_acceptor_service::listen	1189
5.190.18	socket_acceptor_service::local_endpoint	1190
5.190.19	socket_acceptor_service::move_assign	1190
5.190.20	socket_acceptor_service::move_construct	1190
5.190.21	socket_acceptor_service::native	1190
5.190.22	socket_acceptor_service::native_handle	1190
5.190.23	socket_acceptor_service::native_handle_type	1190
5.190.24	socket_acceptor_service::native_non_blocking	1191
5.190.24.1	socket_acceptor_service::native_non_blocking (1 of 2 overloads)	1191
5.190.24.2	socket_acceptor_service::native_non_blocking (2 of 2 overloads)	1191
5.190.25	socket_acceptor_service::native_type	1191
5.190.26	socket_acceptor_service::non_blocking	1191
5.190.26.1	socket_acceptor_service::non_blocking (1 of 2 overloads)	1192
5.190.26.2	socket_acceptor_service::non_blocking (2 of 2 overloads)	1192
5.190.27	socket_acceptor_service::open	1192
5.190.28	socket_acceptor_service::protocol_type	1192
5.190.29	socket_acceptor_service::set_option	1192
5.190.30	socket_acceptor_service::socket_acceptor_service	1192
5.191	socket_base	1193
5.191.1	socket_base::broadcast	1194
5.191.2	socket_base::bytes_readable	1195
5.191.3	socket_base::debug	1195
5.191.4	socket_base::do_not_route	1196
5.191.5	socket_base::enable_connection_aborted	1196
5.191.6	socket_base::keep_alive	1197
5.191.7	socket_base::linger	1197
5.191.8	socket_base::max_connections	1198
5.191.9	socket_base::message_do_not_route	1198
5.191.10	socket_base::message_end_of_record	1198
5.191.11	socket_base::message_flags	1198
5.191.12	socket_base::message_out_of_band	1198
5.191.13	socket_base::message_peek	1199

5.191.14	socket_base::non_blocking_io	1199
5.191.15	socket_base::receive_buffer_size	1199
5.191.16	socket_base::receive_low_watermark	1200
5.191.17	socket_base::reuse_address	1200
5.191.18	socket_base::send_buffer_size	1201
5.191.19	socket_base::send_low_watermark	1201
5.191.20	socket_base::shutdown_type	1202
5.191.21	socket_base::~socket_base	1202
5.192	spawn	1202
5.192.1	spawn (1 of 4 overloads)	1203
5.192.2	spawn (2 of 4 overloads)	1204
5.192.3	spawn (3 of 4 overloads)	1204
5.192.4	spawn (4 of 4 overloads)	1205
5.193	ssl::context	1205
5.193.1	ssl::context::add_certificate_authority	1207
5.193.1.1	ssl::context::add_certificate_authority (1 of 2 overloads)	1207
5.193.1.2	ssl::context::add_certificate_authority (2 of 2 overloads)	1208
5.193.2	ssl::context::add_verify_path	1208
5.193.2.1	ssl::context::add_verify_path (1 of 2 overloads)	1208
5.193.2.2	ssl::context::add_verify_path (2 of 2 overloads)	1209
5.193.3	ssl::context::clear_options	1209
5.193.3.1	ssl::context::clear_options (1 of 2 overloads)	1209
5.193.3.2	ssl::context::clear_options (2 of 2 overloads)	1210
5.193.4	ssl::context::context	1210
5.193.4.1	ssl::context::context (1 of 3 overloads)	1211
5.193.4.2	ssl::context::context (2 of 3 overloads)	1211
5.193.4.3	ssl::context::context (3 of 3 overloads)	1211
5.193.5	ssl::context::default_workarounds	1211
5.193.6	ssl::context::file_format	1211
5.193.7	ssl::context::impl	1212
5.193.8	ssl::context::impl_type	1212
5.193.9	ssl::context::load_verify_file	1212
5.193.9.1	ssl::context::load_verify_file (1 of 2 overloads)	1212
5.193.9.2	ssl::context::load_verify_file (2 of 2 overloads)	1213
5.193.10	ssl::context::method	1213
5.193.11	ssl::context::native_handle	1214
5.193.12	ssl::context::native_handle_type	1214
5.193.13	ssl::context::no_compression	1214
5.193.14	ssl::context::no_sslv2	1214

5.193.15	ssl::context::no_sslv3	1214
5.193.16	ssl::context::no_tls1	1214
5.193.17	ssl::context::no_tls1_1	1214
5.193.18	ssl::context::no_tls1_2	1215
5.193.19	ssl::context::operator=	1215
5.193.20	ssl::context::options	1215
5.193.21	ssl::context::password_purpose	1215
5.193.22	ssl::context::set_default_verify_paths	1216
5.193.22.1	ssl::context::set_default_verify_paths (1 of 2 overloads)	1216
5.193.22.2	ssl::context::set_default_verify_paths (2 of 2 overloads)	1216
5.193.23	ssl::context::set_options	1216
5.193.23.1	ssl::context::set_options (1 of 2 overloads)	1217
5.193.23.2	ssl::context::set_options (2 of 2 overloads)	1217
5.193.24	ssl::context::set_password_callback	1217
5.193.24.1	ssl::context::set_password_callback (1 of 2 overloads)	1218
5.193.24.2	ssl::context::set_password_callback (2 of 2 overloads)	1218
5.193.25	ssl::context::set_verify_callback	1219
5.193.25.1	ssl::context::set_verify_callback (1 of 2 overloads)	1219
5.193.25.2	ssl::context::set_verify_callback (2 of 2 overloads)	1220
5.193.26	ssl::context::set_verify_depth	1220
5.193.26.1	ssl::context::set_verify_depth (1 of 2 overloads)	1220
5.193.26.2	ssl::context::set_verify_depth (2 of 2 overloads)	1221
5.193.27	ssl::context::set_verify_mode	1221
5.193.27.1	ssl::context::set_verify_mode (1 of 2 overloads)	1221
5.193.27.2	ssl::context::set_verify_mode (2 of 2 overloads)	1222
5.193.28	ssl::context::single_dh_use	1222
5.193.29	ssl::context::use_certificate	1222
5.193.29.1	ssl::context::use_certificate (1 of 2 overloads)	1223
5.193.29.2	ssl::context::use_certificate (2 of 2 overloads)	1223
5.193.30	ssl::context::use_certificate_chain	1223
5.193.30.1	ssl::context::use_certificate_chain (1 of 2 overloads)	1224
5.193.30.2	ssl::context::use_certificate_chain (2 of 2 overloads)	1224
5.193.31	ssl::context::use_certificate_chain_file	1224
5.193.31.1	ssl::context::use_certificate_chain_file (1 of 2 overloads)	1225
5.193.31.2	ssl::context::use_certificate_chain_file (2 of 2 overloads)	1225
5.193.32	ssl::context::use_certificate_file	1225
5.193.32.1	ssl::context::use_certificate_file (1 of 2 overloads)	1226
5.193.32.2	ssl::context::use_certificate_file (2 of 2 overloads)	1226
5.193.33	ssl::context::use_private_key	1227

5.193.33.1	ssl::context::use_private_key (1 of 2 overloads)	1227
5.193.33.2	ssl::context::use_private_key (2 of 2 overloads)	1227
5.193.34	ssl::context::use_private_key_file	1228
5.193.34.1	ssl::context::use_private_key_file (1 of 2 overloads)	1228
5.193.34.2	ssl::context::use_private_key_file (2 of 2 overloads)	1228
5.193.35	ssl::context::use_rsa_private_key	1229
5.193.35.1	ssl::context::use_rsa_private_key (1 of 2 overloads)	1229
5.193.35.2	ssl::context::use_rsa_private_key (2 of 2 overloads)	1230
5.193.36	ssl::context::use_rsa_private_key_file	1230
5.193.36.1	ssl::context::use_rsa_private_key_file (1 of 2 overloads)	1230
5.193.36.2	ssl::context::use_rsa_private_key_file (2 of 2 overloads)	1231
5.193.37	ssl::context::use_tmp_dh	1231
5.193.37.1	ssl::context::use_tmp_dh (1 of 2 overloads)	1231
5.193.37.2	ssl::context::use_tmp_dh (2 of 2 overloads)	1232
5.193.38	ssl::context::use_tmp_dh_file	1232
5.193.38.1	ssl::context::use_tmp_dh_file (1 of 2 overloads)	1232
5.193.38.2	ssl::context::use_tmp_dh_file (2 of 2 overloads)	1233
5.193.39	ssl::context::~context	1233
5.194	ssl::context_base	1233
5.194.1	ssl::context_base::default_workarounds	1234
5.194.2	ssl::context_base::file_format	1235
5.194.3	ssl::context_base::method	1235
5.194.4	ssl::context_base::no_compression	1236
5.194.5	ssl::context_base::no_sslv2	1236
5.194.6	ssl::context_base::no_sslv3	1236
5.194.7	ssl::context_base::no_tlsv1	1236
5.194.8	ssl::context_base::no_tlsv1_1	1236
5.194.9	ssl::context_base::no_tlsv1_2	1236
5.194.10	ssl::context_base::options	1236
5.194.11	ssl::context_base::password_purpose	1237
5.194.12	ssl::context_base::single_dh_use	1237
5.194.13	ssl::context_base::~context_base	1237
5.195	ssl::rfc2818_verification	1237
5.195.1	ssl::rfc2818_verification::operator()	1238
5.195.2	ssl::rfc2818_verification::result_type	1238
5.195.3	ssl::rfc2818_verification::rfc2818_verification	1238
5.196	ssl::stream	1239
5.196.1	ssl::stream::async_handshake	1240
5.196.1.1	ssl::stream::async_handshake (1 of 2 overloads)	1241

5.196.1.2	ssl::stream::async_handshake (2 of 2 overloads)	1241
5.196.2	ssl::stream::async_read_some	1242
5.196.3	ssl::stream::async_shutdown	1242
5.196.4	ssl::stream::async_write_some	1243
5.196.5	ssl::stream::get_io_service	1243
5.196.6	ssl::stream::handshake	1243
5.196.6.1	ssl::stream::handshake (1 of 4 overloads)	1244
5.196.6.2	ssl::stream::handshake (2 of 4 overloads)	1244
5.196.6.3	ssl::stream::handshake (3 of 4 overloads)	1245
5.196.6.4	ssl::stream::handshake (4 of 4 overloads)	1245
5.196.7	ssl::stream::handshake_type	1245
5.196.8	ssl::stream::impl	1246
5.196.9	ssl::stream::impl_type	1246
5.196.10	ssl::stream::lowest_layer	1246
5.196.10.1	ssl::stream::lowest_layer (1 of 2 overloads)	1246
5.196.10.2	ssl::stream::lowest_layer (2 of 2 overloads)	1246
5.196.11	ssl::stream::lowest_layer_type	1247
5.196.12	ssl::stream::native_handle	1247
5.196.13	ssl::stream::native_handle_type	1247
5.196.14	ssl::stream::next_layer	1247
5.196.14.1	ssl::stream::next_layer (1 of 2 overloads)	1248
5.196.14.2	ssl::stream::next_layer (2 of 2 overloads)	1248
5.196.15	ssl::stream::next_layer_type	1248
5.196.16	ssl::stream::read_some	1248
5.196.16.1	ssl::stream::read_some (1 of 2 overloads)	1249
5.196.16.2	ssl::stream::read_some (2 of 2 overloads)	1249
5.196.17	ssl::stream::set_verify_callback	1250
5.196.17.1	ssl::stream::set_verify_callback (1 of 2 overloads)	1250
5.196.17.2	ssl::stream::set_verify_callback (2 of 2 overloads)	1251
5.196.18	ssl::stream::set_verify_depth	1251
5.196.18.1	ssl::stream::set_verify_depth (1 of 2 overloads)	1251
5.196.18.2	ssl::stream::set_verify_depth (2 of 2 overloads)	1252
5.196.19	ssl::stream::set_verify_mode	1252
5.196.19.1	ssl::stream::set_verify_mode (1 of 2 overloads)	1252
5.196.19.2	ssl::stream::set_verify_mode (2 of 2 overloads)	1253
5.196.20	ssl::stream::shutdown	1253
5.196.20.1	ssl::stream::shutdown (1 of 2 overloads)	1253
5.196.20.2	ssl::stream::shutdown (2 of 2 overloads)	1254
5.196.21	ssl::stream::stream	1254

5.196.22	ssl::stream::write_some	1254
5.196.22.1	ssl::stream::write_some (1 of 2 overloads)	1254
5.196.22.2	ssl::stream::write_some (2 of 2 overloads)	1255
5.196.23	ssl::stream::~stream	1255
5.197	ssl::stream::impl_struct	1256
5.197.1	ssl::stream::impl_struct::ssl	1256
5.198	ssl::stream_base	1256
5.198.1	ssl::stream_base::handshake_type	1257
5.198.2	ssl::stream_base::~stream_base	1257
5.199	ssl::verify_client_once	1257
5.200	ssl::verify_context	1257
5.200.1	ssl::verify_context::native_handle	1258
5.200.2	ssl::verify_context::native_handle_type	1258
5.200.3	ssl::verify_context::verify_context	1258
5.201	ssl::verify_fail_if_no_peer_cert	1258
5.202	ssl::verify_mode	1259
5.203	ssl::verify_none	1259
5.204	ssl::verify_peer	1259
5.205	steady_timer	1259
5.206	strand	1262
5.207	stream_socket_service	1264
5.207.1	stream_socket_service::assign	1266
5.207.2	stream_socket_service::async_connect	1266
5.207.3	stream_socket_service::async_receive	1266
5.207.4	stream_socket_service::async_send	1267
5.207.5	stream_socket_service::at_mark	1267
5.207.6	stream_socket_service::available	1267
5.207.7	stream_socket_service::bind	1267
5.207.8	stream_socket_service::cancel	1267
5.207.9	stream_socket_service::close	1267
5.207.10	stream_socket_service::connect	1268
5.207.11	stream_socket_service::construct	1268
5.207.12	stream_socket_service::converting_move_construct	1268
5.207.13	stream_socket_service::destroy	1268
5.207.14	stream_socket_service::endpoint_type	1268
5.207.15	stream_socket_service::get_io_service	1268
5.207.16	stream_socket_service::get_option	1269
5.207.17	stream_socket_service::id	1269
5.207.18	stream_socket_service::implementation_type	1269

5.207.19	stream_socket_service::io_control	1269
5.207.20	stream_socket_service::is_open	1269
5.207.21	stream_socket_service::local_endpoint	1269
5.207.22	stream_socket_service::move_assign	1270
5.207.23	stream_socket_service::move_construct	1270
5.207.24	stream_socket_service::native	1270
5.207.25	stream_socket_service::native_handle	1270
5.207.26	stream_socket_service::native_handle_type	1270
5.207.27	stream_socket_service::native_non_blocking	1270
5.207.27.1	stream_socket_service::native_non_blocking (1 of 2 overloads)	1271
5.207.27.2	stream_socket_service::native_non_blocking (2 of 2 overloads)	1271
5.207.28	stream_socket_service::native_type	1271
5.207.29	stream_socket_service::non_blocking	1271
5.207.29.1	stream_socket_service::non_blocking (1 of 2 overloads)	1271
5.207.29.2	stream_socket_service::non_blocking (2 of 2 overloads)	1272
5.207.30	stream_socket_service::open	1272
5.207.31	stream_socket_service::protocol_type	1272
5.207.32	stream_socket_service::receive	1272
5.207.33	stream_socket_service::remote_endpoint	1272
5.207.34	stream_socket_service::send	1273
5.207.35	stream_socket_service::set_option	1273
5.207.36	stream_socket_service::shutdown	1273
5.207.37	stream_socket_service::stream_socket_service	1273
5.208	streambuf	1273
5.209	system_category	1275
5.210	system_error	1275
5.210.1	system_error::code	1276
5.210.2	system_error::operator=	1276
5.210.3	system_error::system_error	1276
5.210.3.1	system_error::system_error (1 of 3 overloads)	1277
5.210.3.2	system_error::system_error (2 of 3 overloads)	1277
5.210.3.3	system_error::system_error (3 of 3 overloads)	1277
5.210.4	system_error::what	1277
5.210.5	system_error::~system_error	1277
5.211	system_timer	1277
5.212	thread	1280
5.212.1	thread::join	1281
5.212.2	thread::thread	1281
5.212.3	thread::~thread	1281

5.213	time_traits< boost::posix_time::ptime >	1281
5.213.1	time_traits< boost::posix_time::ptime >::add	1283
5.213.2	time_traits< boost::posix_time::ptime >::duration_type	1283
5.213.3	time_traits< boost::posix_time::ptime >::less_than	1283
5.213.4	time_traits< boost::posix_time::ptime >::now	1284
5.213.5	time_traits< boost::posix_time::ptime >::subtract	1284
5.213.6	time_traits< boost::posix_time::ptime >::time_type	1284
5.213.7	time_traits< boost::posix_time::ptime >::to_posix_duration	1284
5.214	transfer_all	1284
5.215	transfer_at_least	1285
5.216	transfer_exactly	1285
5.217	use_future	1286
5.218	use_future_t	1286
5.218.1	use_future_t::allocator_type	1287
5.218.2	use_future_t::get_allocator	1287
5.218.3	use_future_t::operator[]	1287
5.218.4	use_future_t::use_future_t	1288
5.218.4.1	use_future_t::use_future_t (1 of 2 overloads)	1288
5.218.4.2	use_future_t::use_future_t (2 of 2 overloads)	1288
5.219	use_service	1288
5.220	wait_traits	1289
5.220.1	wait_traits::to_wait_duration	1289
5.221	waitable_timer_service	1289
5.221.1	waitable_timer_service::async_wait	1290
5.221.2	waitable_timer_service::cancel	1291
5.221.3	waitable_timer_service::cancel_one	1291
5.221.4	waitable_timer_service::clock_type	1291
5.221.5	waitable_timer_service::construct	1291
5.221.6	waitable_timer_service::destroy	1291
5.221.7	waitable_timer_service::duration	1291
5.221.8	waitable_timer_service::expires_at	1292
5.221.8.1	waitable_timer_service::expires_at (1 of 2 overloads)	1292
5.221.8.2	waitable_timer_service::expires_at (2 of 2 overloads)	1292
5.221.9	waitable_timer_service::expires_from_now	1292
5.221.9.1	waitable_timer_service::expires_from_now (1 of 2 overloads)	1292
5.221.9.2	waitable_timer_service::expires_from_now (2 of 2 overloads)	1293
5.221.10	waitable_timer_service::get_io_service	1293
5.221.11	waitable_timer_service::id	1293
5.221.12	waitable_timer_service::implementation_type	1293

5.221.13	waitable_timer_service::time_point	1293
5.221.14	waitable_timer_service::traits_type	1293
5.221.15	waitable_timer_service::wait	1294
5.221.16	waitable_timer_service::waitable_timer_service	1294
5.222	windows::basic_handle	1294
5.222.1	windows::basic_handle::assign	1296
5.222.1.1	windows::basic_handle::assign (1 of 2 overloads)	1296
5.222.1.2	windows::basic_handle::assign (2 of 2 overloads)	1296
5.222.2	windows::basic_handle::basic_handle	1296
5.222.2.1	windows::basic_handle::basic_handle (1 of 3 overloads)	1297
5.222.2.2	windows::basic_handle::basic_handle (2 of 3 overloads)	1297
5.222.2.3	windows::basic_handle::basic_handle (3 of 3 overloads)	1297
5.222.3	windows::basic_handle::cancel	1298
5.222.3.1	windows::basic_handle::cancel (1 of 2 overloads)	1298
5.222.3.2	windows::basic_handle::cancel (2 of 2 overloads)	1298
5.222.4	windows::basic_handle::close	1298
5.222.4.1	windows::basic_handle::close (1 of 2 overloads)	1298
5.222.4.2	windows::basic_handle::close (2 of 2 overloads)	1299
5.222.5	windows::basic_handle::get_implementation	1299
5.222.5.1	windows::basic_handle::get_implementation (1 of 2 overloads)	1299
5.222.5.2	windows::basic_handle::get_implementation (2 of 2 overloads)	1299
5.222.6	windows::basic_handle::get_io_service	1299
5.222.7	windows::basic_handle::get_service	1300
5.222.7.1	windows::basic_handle::get_service (1 of 2 overloads)	1300
5.222.7.2	windows::basic_handle::get_service (2 of 2 overloads)	1300
5.222.8	windows::basic_handle::implementation	1300
5.222.9	windows::basic_handle::implementation_type	1300
5.222.10	windows::basic_handle::is_open	1300
5.222.11	windows::basic_handle::lowest_layer	1301
5.222.11.1	windows::basic_handle::lowest_layer (1 of 2 overloads)	1301
5.222.11.2	windows::basic_handle::lowest_layer (2 of 2 overloads)	1301
5.222.12	windows::basic_handle::lowest_layer_type	1301
5.222.13	windows::basic_handle::native	1303
5.222.14	windows::basic_handle::native_handle	1303
5.222.15	windows::basic_handle::native_handle_type	1303
5.222.16	windows::basic_handle::native_type	1304
5.222.17	windows::basic_handle::operator=	1304
5.222.18	windows::basic_handle::service	1304
5.222.19	windows::basic_handle::service_type	1304

5.222.20	windows::basic_handle::~basic_handle	1305
5.223	windows::basic_object_handle	1305
5.223.1	windows::basic_object_handle::assign	1306
5.223.1.1	windows::basic_object_handle::assign (1 of 2 overloads)	1307
5.223.1.2	windows::basic_object_handle::assign (2 of 2 overloads)	1307
5.223.2	windows::basic_object_handle::async_wait	1307
5.223.3	windows::basic_object_handle::basic_object_handle	1308
5.223.3.1	windows::basic_object_handle::basic_object_handle (1 of 3 overloads)	1308
5.223.3.2	windows::basic_object_handle::basic_object_handle (2 of 3 overloads)	1308
5.223.3.3	windows::basic_object_handle::basic_object_handle (3 of 3 overloads)	1309
5.223.4	windows::basic_object_handle::cancel	1309
5.223.4.1	windows::basic_object_handle::cancel (1 of 2 overloads)	1309
5.223.4.2	windows::basic_object_handle::cancel (2 of 2 overloads)	1309
5.223.5	windows::basic_object_handle::close	1310
5.223.5.1	windows::basic_object_handle::close (1 of 2 overloads)	1310
5.223.5.2	windows::basic_object_handle::close (2 of 2 overloads)	1310
5.223.6	windows::basic_object_handle::get_implementation	1310
5.223.6.1	windows::basic_object_handle::get_implementation (1 of 2 overloads)	1311
5.223.6.2	windows::basic_object_handle::get_implementation (2 of 2 overloads)	1311
5.223.7	windows::basic_object_handle::get_io_service	1311
5.223.8	windows::basic_object_handle::get_service	1311
5.223.8.1	windows::basic_object_handle::get_service (1 of 2 overloads)	1311
5.223.8.2	windows::basic_object_handle::get_service (2 of 2 overloads)	1311
5.223.9	windows::basic_object_handle::implementation	1312
5.223.10	windows::basic_object_handle::implementation_type	1312
5.223.11	windows::basic_object_handle::is_open	1312
5.223.12	windows::basic_object_handle::lowest_layer	1312
5.223.12.1	windows::basic_object_handle::lowest_layer (1 of 2 overloads)	1312
5.223.12.2	windows::basic_object_handle::lowest_layer (2 of 2 overloads)	1313
5.223.13	windows::basic_object_handle::lowest_layer_type	1313
5.223.14	windows::basic_object_handle::native	1315
5.223.15	windows::basic_object_handle::native_handle	1315
5.223.16	windows::basic_object_handle::native_handle_type	1315
5.223.17	windows::basic_object_handle::native_type	1315
5.223.18	windows::basic_object_handle::operator=	1315
5.223.19	windows::basic_object_handle::service	1316
5.223.20	windows::basic_object_handle::service_type	1316
5.223.21	windows::basic_object_handle::wait	1316
5.223.21.1	windows::basic_object_handle::wait (1 of 2 overloads)	1316

5.223.21.2	windows::basic_object_handle::wait (2 of 2 overloads)	1317
5.224	windows::basic_random_access_handle	1317
5.224.1	windows::basic_random_access_handle::assign	1319
5.224.1.1	windows::basic_random_access_handle::assign (1 of 2 overloads)	1319
5.224.1.2	windows::basic_random_access_handle::assign (2 of 2 overloads)	1319
5.224.2	windows::basic_random_access_handle::async_read_some_at	1319
5.224.3	windows::basic_random_access_handle::async_write_some_at	1320
5.224.4	windows::basic_random_access_handle::basic_random_access_handle	1321
5.224.4.1	windows::basic_random_access_handle::basic_random_access_handle (1 of 3 overloads)	1321
5.224.4.2	windows::basic_random_access_handle::basic_random_access_handle (2 of 3 overloads)	1322
5.224.4.3	windows::basic_random_access_handle::basic_random_access_handle (3 of 3 overloads)	1322
5.224.5	windows::basic_random_access_handle::cancel	1322
5.224.5.1	windows::basic_random_access_handle::cancel (1 of 2 overloads)	1323
5.224.5.2	windows::basic_random_access_handle::cancel (2 of 2 overloads)	1323
5.224.6	windows::basic_random_access_handle::close	1323
5.224.6.1	windows::basic_random_access_handle::close (1 of 2 overloads)	1323
5.224.6.2	windows::basic_random_access_handle::close (2 of 2 overloads)	1324
5.224.7	windows::basic_random_access_handle::get_implementation	1324
5.224.7.1	windows::basic_random_access_handle::get_implementation (1 of 2 overloads)	1324
5.224.7.2	windows::basic_random_access_handle::get_implementation (2 of 2 overloads)	1324
5.224.8	windows::basic_random_access_handle::get_io_service	1324
5.224.9	windows::basic_random_access_handle::get_service	1325
5.224.9.1	windows::basic_random_access_handle::get_service (1 of 2 overloads)	1325
5.224.9.2	windows::basic_random_access_handle::get_service (2 of 2 overloads)	1325
5.224.10	windows::basic_random_access_handle::implementation	1325
5.224.11	windows::basic_random_access_handle::implementation_type	1325
5.224.12	windows::basic_random_access_handle::is_open	1325
5.224.13	windows::basic_random_access_handle::lowest_layer	1326
5.224.13.1	windows::basic_random_access_handle::lowest_layer (1 of 2 overloads)	1326
5.224.13.2	windows::basic_random_access_handle::lowest_layer (2 of 2 overloads)	1326
5.224.14	windows::basic_random_access_handle::lowest_layer_type	1326
5.224.15	windows::basic_random_access_handle::native	1328
5.224.16	windows::basic_random_access_handle::native_handle	1328
5.224.17	windows::basic_random_access_handle::native_handle_type	1329
5.224.18	windows::basic_random_access_handle::native_type	1329
5.224.19	windows::basic_random_access_handle::operator=	1329
5.224.20	windows::basic_random_access_handle::read_some_at	1329
5.224.20.1	windows::basic_random_access_handle::read_some_at (1 of 2 overloads)	1330
5.224.20.2	windows::basic_random_access_handle::read_some_at (2 of 2 overloads)	1331

5.224.21	windows::basic_random_access_handle::service	1331
5.224.22	windows::basic_random_access_handle::service_type	1331
5.224.23	windows::basic_random_access_handle::write_some_at	1332
5.224.23.1	windows::basic_random_access_handle::write_some_at (1 of 2 overloads)	1332
5.224.23.2	windows::basic_random_access_handle::write_some_at (2 of 2 overloads)	1333
5.225	windows::basic_stream_handle	1333
5.225.1	windows::basic_stream_handle::assign	1335
5.225.1.1	windows::basic_stream_handle::assign (1 of 2 overloads)	1335
5.225.1.2	windows::basic_stream_handle::assign (2 of 2 overloads)	1336
5.225.2	windows::basic_stream_handle::async_read_some	1336
5.225.3	windows::basic_stream_handle::async_write_some	1337
5.225.4	windows::basic_stream_handle::basic_stream_handle	1337
5.225.4.1	windows::basic_stream_handle::basic_stream_handle (1 of 3 overloads)	1338
5.225.4.2	windows::basic_stream_handle::basic_stream_handle (2 of 3 overloads)	1338
5.225.4.3	windows::basic_stream_handle::basic_stream_handle (3 of 3 overloads)	1338
5.225.5	windows::basic_stream_handle::cancel	1339
5.225.5.1	windows::basic_stream_handle::cancel (1 of 2 overloads)	1339
5.225.5.2	windows::basic_stream_handle::cancel (2 of 2 overloads)	1339
5.225.6	windows::basic_stream_handle::close	1339
5.225.6.1	windows::basic_stream_handle::close (1 of 2 overloads)	1340
5.225.6.2	windows::basic_stream_handle::close (2 of 2 overloads)	1340
5.225.7	windows::basic_stream_handle::get_implementation	1340
5.225.7.1	windows::basic_stream_handle::get_implementation (1 of 2 overloads)	1340
5.225.7.2	windows::basic_stream_handle::get_implementation (2 of 2 overloads)	1340
5.225.8	windows::basic_stream_handle::get_io_service	1341
5.225.9	windows::basic_stream_handle::get_service	1341
5.225.9.1	windows::basic_stream_handle::get_service (1 of 2 overloads)	1341
5.225.9.2	windows::basic_stream_handle::get_service (2 of 2 overloads)	1341
5.225.10	windows::basic_stream_handle::implementation	1341
5.225.11	windows::basic_stream_handle::implementation_type	1341
5.225.12	windows::basic_stream_handle::is_open	1342
5.225.13	windows::basic_stream_handle::lowest_layer	1342
5.225.13.1	windows::basic_stream_handle::lowest_layer (1 of 2 overloads)	1342
5.225.13.2	windows::basic_stream_handle::lowest_layer (2 of 2 overloads)	1342
5.225.14	windows::basic_stream_handle::lowest_layer_type	1343
5.225.15	windows::basic_stream_handle::native	1344
5.225.16	windows::basic_stream_handle::native_handle	1344
5.225.17	windows::basic_stream_handle::native_handle_type	1345
5.225.18	windows::basic_stream_handle::native_type	1345

5.225.19	windows::basic_stream_handle::operator=	1345
5.225.20	windows::basic_stream_handle::read_some	1345
5.225.20.1	windows::basic_stream_handle::read_some (1 of 2 overloads)	1346
5.225.20.2	windows::basic_stream_handle::read_some (2 of 2 overloads)	1346
5.225.21	windows::basic_stream_handle::service	1347
5.225.22	windows::basic_stream_handle::service_type	1347
5.225.23	windows::basic_stream_handle::write_some	1347
5.225.23.1	windows::basic_stream_handle::write_some (1 of 2 overloads)	1348
5.225.23.2	windows::basic_stream_handle::write_some (2 of 2 overloads)	1348
5.226	windows::object_handle	1349
5.227	windows::object_handle_service	1351
5.227.1	windows::object_handle_service::assign	1352
5.227.2	windows::object_handle_service::async_wait	1352
5.227.3	windows::object_handle_service::cancel	1352
5.227.4	windows::object_handle_service::close	1352
5.227.5	windows::object_handle_service::construct	1352
5.227.6	windows::object_handle_service::destroy	1353
5.227.7	windows::object_handle_service::get_io_service	1353
5.227.8	windows::object_handle_service::id	1353
5.227.9	windows::object_handle_service::implementation_type	1353
5.227.10	windows::object_handle_service::is_open	1353
5.227.11	windows::object_handle_service::move_assign	1353
5.227.12	windows::object_handle_service::move_construct	1354
5.227.13	windows::object_handle_service::native_handle	1354
5.227.14	windows::object_handle_service::native_handle_type	1354
5.227.15	windows::object_handle_service::object_handle_service	1354
5.227.16	windows::object_handle_service::wait	1354
5.228	windows::overlapped_ptr	1354
5.228.1	windows::overlapped_ptr::complete	1355
5.228.2	windows::overlapped_ptr::get	1355
5.228.2.1	windows::overlapped_ptr::get (1 of 2 overloads)	1356
5.228.2.2	windows::overlapped_ptr::get (2 of 2 overloads)	1356
5.228.3	windows::overlapped_ptr::overlapped_ptr	1356
5.228.3.1	windows::overlapped_ptr::overlapped_ptr (1 of 2 overloads)	1356
5.228.3.2	windows::overlapped_ptr::overlapped_ptr (2 of 2 overloads)	1356
5.228.4	windows::overlapped_ptr::release	1356
5.228.5	windows::overlapped_ptr::reset	1357
5.228.5.1	windows::overlapped_ptr::reset (1 of 2 overloads)	1357
5.228.5.2	windows::overlapped_ptr::reset (2 of 2 overloads)	1357

5.228.6	windows::overlapped_ptr::~overlapped_ptr	1357
5.229	windows::random_access_handle	1357
5.230	windows::random_access_handle_service	1359
5.230.1	windows::random_access_handle_service::assign	1361
5.230.2	windows::random_access_handle_service::async_read_some_at	1361
5.230.3	windows::random_access_handle_service::async_write_some_at	1361
5.230.4	windows::random_access_handle_service::cancel	1361
5.230.5	windows::random_access_handle_service::close	1361
5.230.6	windows::random_access_handle_service::construct	1362
5.230.7	windows::random_access_handle_service::destroy	1362
5.230.8	windows::random_access_handle_service::get_io_service	1362
5.230.9	windows::random_access_handle_service::id	1362
5.230.10	windows::random_access_handle_service::implementation_type	1362
5.230.11	windows::random_access_handle_service::is_open	1362
5.230.12	windows::random_access_handle_service::move_assign	1362
5.230.13	windows::random_access_handle_service::move_construct	1363
5.230.14	windows::random_access_handle_service::native	1363
5.230.15	windows::random_access_handle_service::native_handle	1363
5.230.16	windows::random_access_handle_service::native_handle_type	1363
5.230.17	windows::random_access_handle_service::native_type	1363
5.230.18	windows::random_access_handle_service::random_access_handle_service	1363
5.230.19	windows::random_access_handle_service::read_some_at	1364
5.230.20	windows::random_access_handle_service::write_some_at	1364
5.231	windows::stream_handle	1364
5.232	windows::stream_handle_service	1366
5.232.1	windows::stream_handle_service::assign	1367
5.232.2	windows::stream_handle_service::async_read_some	1367
5.232.3	windows::stream_handle_service::async_write_some	1368
5.232.4	windows::stream_handle_service::cancel	1368
5.232.5	windows::stream_handle_service::close	1368
5.232.6	windows::stream_handle_service::construct	1368
5.232.7	windows::stream_handle_service::destroy	1368
5.232.8	windows::stream_handle_service::get_io_service	1368
5.232.9	windows::stream_handle_service::id	1369
5.232.10	windows::stream_handle_service::implementation_type	1369
5.232.11	windows::stream_handle_service::is_open	1369
5.232.12	windows::stream_handle_service::move_assign	1369
5.232.13	windows::stream_handle_service::move_construct	1369
5.232.14	windows::stream_handle_service::native	1369

5.232.15	windows::stream_handle_service::native_handle	1370
5.232.16	windows::stream_handle_service::native_handle_type	1370
5.232.17	windows::stream_handle_service::native_type	1370
5.232.18	windows::stream_handle_service::read_some	1370
5.232.19	windows::stream_handle_service::stream_handle_service	1370
5.232.20	windows::stream_handle_service::write_some	1371
5.233	write	1371
5.233.1	write (1 of 8 overloads)	1372
5.233.2	write (2 of 8 overloads)	1373
5.233.3	write (3 of 8 overloads)	1374
5.233.4	write (4 of 8 overloads)	1375
5.233.5	write (5 of 8 overloads)	1376
5.233.6	write (6 of 8 overloads)	1377
5.233.7	write (7 of 8 overloads)	1378
5.233.8	write (8 of 8 overloads)	1379
5.234	write_at	1379
5.234.1	write_at (1 of 8 overloads)	1381
5.234.2	write_at (2 of 8 overloads)	1382
5.234.3	write_at (3 of 8 overloads)	1383
5.234.4	write_at (4 of 8 overloads)	1384
5.234.5	write_at (5 of 8 overloads)	1385
5.234.6	write_at (6 of 8 overloads)	1386
5.234.7	write_at (7 of 8 overloads)	1387
5.234.8	write_at (8 of 8 overloads)	1388
5.235	yield_context	1389

6 Revision History

1390

List of Tables

1	Buffer-oriented asynchronous random-access read device requirements	169
2	Buffer-oriented asynchronous random-access write device requirements	171
3	Buffer-oriented asynchronous read stream requirements	172
4	Buffer-oriented asynchronous write stream requirements	173
5	ConstBufferSequence requirements	176
6	ConvertibleToConstBuffer requirements	179
7	ConvertibleToMutableBuffer requirements	180
8	DatagramSocketService requirements	181
9	DescriptorService requirements	187
10	Endpoint requirements	188
11	GettableSerialPortOption requirements	189
12	GettableSocketOption requirements	190
13	Handler requirements	191
14	HandleService requirements	192
15	InternetProtocol requirements	195
16	IoControlCommand requirements	195
17	IoObjectService requirements	195
18	MutableBufferSequence requirements	196
19	ObjectHandleService requirements	199
20	Protocol requirements	199
21	RandomAccessHandleService requirements	200
22	RawSocketService requirements	204
23	ResolverService requirements	211
24	StreamSocketService requirements	213
25	SerialPortService requirements	217
26	SettableSerialPortOption requirements	222
27	SettableSocketOption requirements	222
28	SignalSetService requirements	225
29	SocketAcceptorService requirements	226
30	SocketService requirements	229
31	StreamDescriptorService requirements	232
32	StreamHandleService requirements	236
33	StreamSocketService requirements	240
34	Buffer-oriented synchronous random-access read device requirements	244
35	Buffer-oriented synchronous random-access write device requirements	244
36	Buffer-oriented synchronous read stream requirements	245
37	Buffer-oriented synchronous write stream requirements	246
38	TimeTraits requirements	247
39	TimerService requirements	248
40	WaitableTimerService requirements	249
41	WaitTraits requirements	251

1 Overview

- Rationale
- Core Concepts and Functionality
 - Basic Asio Anatomy
 - The Proactor Design Pattern: Concurrency Without Threads
 - Threads and Asio
 - Strands: Use Threads Without Explicit Locking
 - Buffers
 - Streams, Short Reads and Short Writes
 - Reactor-Style Operations
 - Line-Based Operations
 - Custom Memory Allocation
 - Handler Tracking
 - Stackless Coroutines
 - Stackful Coroutines
- Networking
 - TCP, UDP and ICMP
 - Support for Other Protocols
 - Socket Iostreams
 - The BSD Socket API and Asio
- Timers
- Serial Ports
- Signal Handling
- POSIX-Specific Functionality
 - UNIX Domain Sockets
 - Stream-Oriented File Descriptors
- Windows-Specific Functionality
 - Stream-Oriented HANDLEs
 - Random-Access HANDLEs
- SSL
- C++ 2011 Support
- Platform-Specific Implementation Notes

1.1 Rationale

Most programs interact with the outside world in some way, whether it be via a file, a network, a serial cable, or the console. Sometimes, as is the case with networking, individual I/O operations can take a long time to complete. This poses particular challenges to application development.

Asio provides the tools to manage these long running operations, without requiring programs to use concurrency models based on threads and explicit locking.

The Asio library is intended for programmers using C++ for systems programming, where access to operating system functionality such as networking is often required. In particular, Asio addresses the following goals:

- **Portability.** The library should support a range of commonly used operating systems, and provide consistent behaviour across these operating systems.
- **Scalability.** The library should facilitate the development of network applications that scale to thousands of concurrent connections. The library implementation for each operating system should use the mechanism that best enables this scalability.
- **Efficiency.** The library should support techniques such as scatter-gather I/O, and allow programs to minimise data copying.
- **Model concepts from established APIs, such as BSD sockets.** The BSD socket API is widely implemented and understood, and is covered in much literature. Other programming languages often use a similar interface for networking APIs. As far as is reasonable, Asio should leverage existing practice.
- **Ease of use.** The library should provide a lower entry barrier for new users by taking a toolkit, rather than framework, approach. That is, it should try to minimise the up-front investment in time to just learning a few basic rules and guidelines. After that, a library user should only need to understand the specific functions that are being used.
- **Basis for further abstraction.** The library should permit the development of other libraries that provide higher levels of abstraction. For example, implementations of commonly used protocols such as HTTP.

Although Asio started life focused primarily on networking, its concepts of asynchronous I/O have been extended to include other operating system resources such as serial ports, file descriptors, and so on.

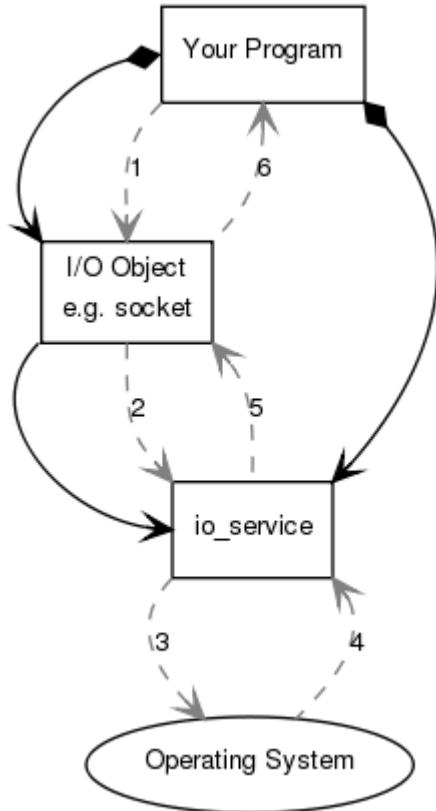
1.2 Core Concepts and Functionality

- Basic Asio Anatomy
- The Proactor Design Pattern: Concurrency Without Threads
- Threads and Asio
- Strands: Use Threads Without Explicit Locking
- Buffers
- Streams, Short Reads and Short Writes
- Reactor-Style Operations
- Line-Based Operations
- Custom Memory Allocation
- Handler Tracking
- Stackless Coroutines
- Stackful Coroutines

1.2.1 Basic Asio Anatomy

Asio may be used to perform both synchronous and asynchronous operations on I/O objects such as sockets. Before using Asio it may be useful to get a conceptual picture of the various parts of Asio, your program, and how they work together.

As an introductory example, let's consider what happens when you perform a connect operation on a socket. We shall start by examining synchronous operations.



Your program will have at least one **io_service** object. The **io_service** represents your program's link to the operating system's I/O services.

```
asio::io_service io_service;
```

To perform I/O operations your program will need an **I/O object** such as a TCP socket:

```
asio::ip::tcp::socket socket(io_service);
```

When a synchronous connect operation is performed, the following sequence of events occurs:

1. Your program initiates the connect operation by calling the **I/O object**:

```
socket.connect(server_endpoint);
```

2. The **I/O object** forwards the request to the **io_service**.

3. The **io_service** calls on the **operating system** to perform the connect operation.

4. The **operating system** returns the result of the operation to the **io_service**.

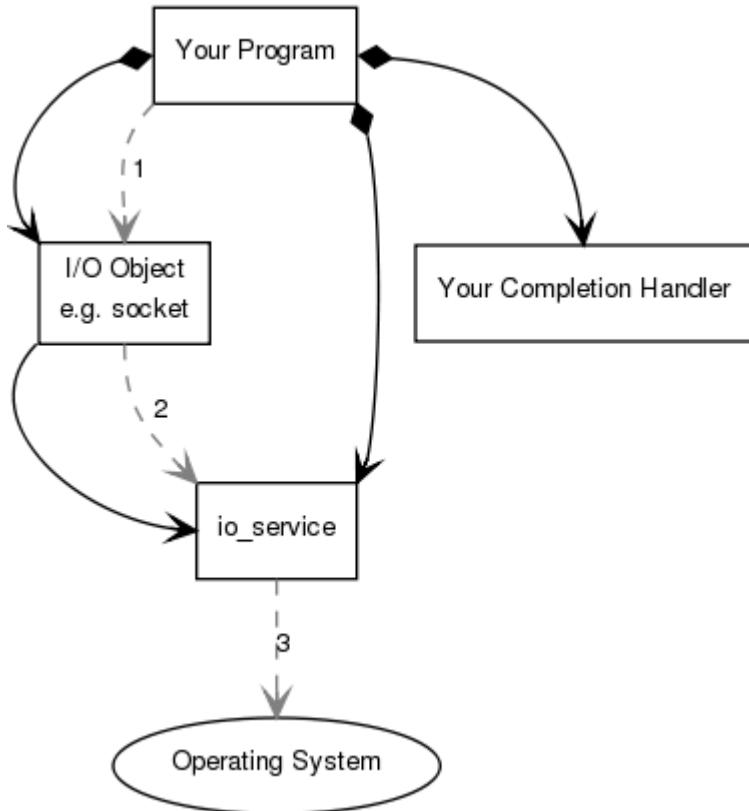
5. The **io_service** translates any error resulting from the operation into an object of type `asio::error_code`. An `error_code` may be compared with specific values, or tested as a boolean (where a `false` result means that no error occurred). The result is then forwarded back up to the **I/O object**.

6. The **I/O object** throws an exception of type `asio::system_error` if the operation failed. If the code to initiate the operation had instead been written as:

```
asio::error_code ec;
socket.connect(server_endpoint, ec);
```

then the `error_code` variable `ec` would be set to the result of the operation, and no exception would be thrown.

When an asynchronous operation is used, a different sequence of events occurs.



1. Your program initiates the connect operation by calling the **I/O object**:

```
socket.async_connect(server_endpoint, your_completion_handler);
```

where `your_completion_handler` is a function or function object with the signature:

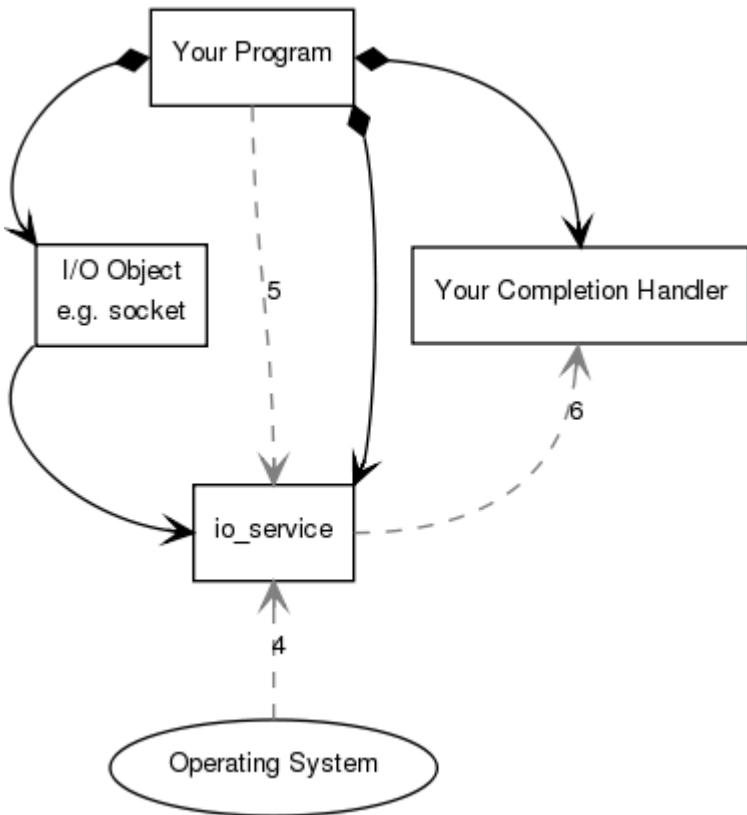
```
void your_completion_handler(const asio::error_code& ec);
```

The exact signature required depends on the asynchronous operation being performed. The reference documentation indicates the appropriate form for each operation.

2. The **I/O object** forwards the request to the **io_service**.

3. The **io_service** signals to the **operating system** that it should start an asynchronous connect.

Time passes. (In the synchronous case this wait would have been contained entirely within the duration of the connect operation.)



4. The **operating system** indicates that the connect operation has completed by placing the result on a queue, ready to be picked up by the **io_service**.
5. **Your program** must make a call to `io_service::run()` (or to one of the similar **io_service** member functions) in order for the result to be retrieved. A call to `io_service::run()` blocks while there are unfinished asynchronous operations, so you would typically call it as soon as you have started your first asynchronous operation.
6. While inside the call to `io_service::run()`, the **io_service** dequeues the result of the operation, translates it into an `error_code`, and then passes it to **your completion handler**.

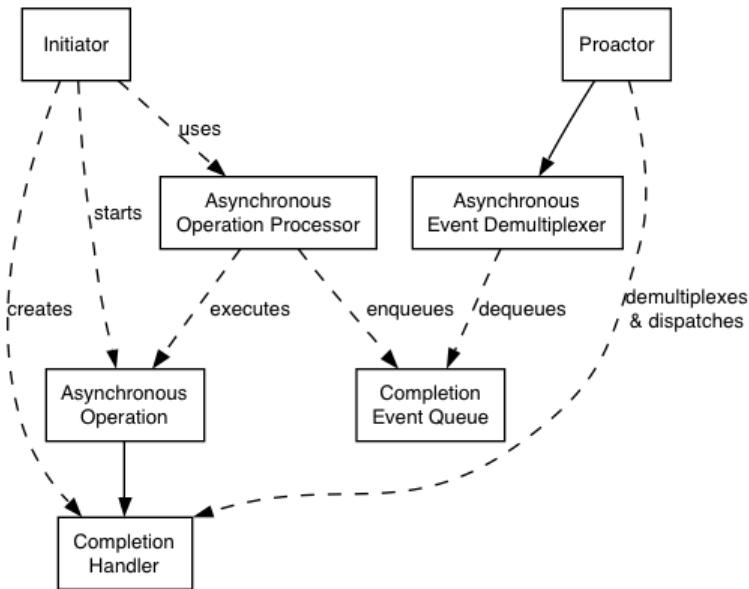
This is a simplified picture of how Asio operates. You will want to delve further into the documentation if your needs are more advanced, such as extending Asio to perform other types of asynchronous operations.

1.2.2 The Proactor Design Pattern: Concurrency Without Threads

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

Proactor and Asio

Let us examine how the Proactor design pattern is implemented in Asio, without reference to platform-specific details.



Proactor design pattern (adapted from [POSA2])

— Asynchronous Operation

— Asynchronous Operation Processor

— Completion Event Queue

— Completion Handler

— Asynchronous Event Demultiplexer

— Proactor

— Initiator

Implementation Using Reactor

On many platforms, Asio implements the Proactor design pattern in terms of a Reactor, such as `select`, `epoll` or `kqueue`. This implementation approach corresponds to the Proactor design pattern as follows:

- Asynchronous Operation Processor

- Completion Event Queue

- Asynchronous Event Demultiplexer

Implementation Using Windows Overlapped I/O

On Windows NT, 2000 and XP, Asio takes advantage of overlapped I/O to provide an efficient implementation of the Proactor design pattern. This implementation approach corresponds to the Proactor design pattern as follows:

- Asynchronous Operation Processor

- Completion Event Queue

- Asynchronous Event Demultiplexer

Advantages

- Portability.

- Decoupling threading from concurrency.

- Performance and scalability.

- Simplified application synchronisation.

- Function composition.

Disadvantages

— Program complexity.

— Memory usage.

References

[POSA2] D. Schmidt et al, *Pattern Oriented Software Architecture, Volume 2*. Wiley, 2000.

1.2.3 Threads and Asio

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.

Thread Pools

Multiple threads may call `io_service::run()` to set up a pool of threads from which completion handlers may be invoked. This approach may also be used with `io_service::post()` to use a means to perform any computational tasks across a thread pool.

Note that all threads that have joined an `io_service`'s pool are considered equivalent, and the `io_service` may distribute work across them in an arbitrary fashion.

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.

See Also

[io_service](#).

1.2.4 Strands: Use Threads Without Explicit Locking

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 asio_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 `asio_handler_invoke` so that the function object is executed through the strand.

See Also

[io_service::strand](#), [tutorial Timer.5, HTTP server 3 example](#).

1.2.5 Buffers

Fundamentally, I/O involves the transfer of data to and from contiguous regions of memory, called buffers. These buffers can be simply expressed as a tuple consisting of a pointer and a size in bytes. However, to allow the development of efficient network applications, Asio includes support for scatter-gather operations. These operations involve one or more buffers:

- 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 Asio 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.

In addition to specifying buffers as a pointer and size in bytes, Asio makes a distinction between modifiable memory (called `mutable`) 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, Asio does not use the above definitions as-is, 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, `boost::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 `read()` or `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 `boost::array` can be used.

Streambuf for Integration with Iostreams

The class `asio::basic_streambuf` is derived from `std::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 `streambuf` 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:

- The input sequence of the `streambuf` is accessible via the `data()` member function. The return type of this function meets the `ConstBufferSequence` requirements.
- The output sequence of the `streambuf` is accessible via the `prepare()` member function. The return type of this function meets the `MutableBufferSequence` requirements.
- Data is transferred from the front of the output sequence to the back of the input sequence by calling the `commit()` member function.
- Data is removed from the front of the input sequence by calling the `consume()` member function.

The `streambuf` constructor accepts a `size_t` argument specifying the maximum of the sum of the sizes of the input sequence and output sequence. Any operation that would, if successful, grow the internal data beyond this limit will throw a `std::length_error` exception.

Bytewise Traversal of Buffer Sequences

The `buffers_iterator<>` class template allows buffer sequences (i.e. types meeting `MutableBufferSequence` or `ConstBufferSequence` requirements) to be traversed as though they were a contiguous sequence of bytes. Helper functions called `buffers_begin()` and `buffers_end()` are also provided, where the `buffers_iterator<>` template parameter is automatically deduced.

As an example, to read a single line from a socket and into a `std::string`, you may write:

```
asio::streambuf sb;
...
std::size_t n = asio::read_until(sock, sb, '\n');
asio::streambuf::const_buffers_type bufs = sb.data();
std::string line(
    asio::buffers_begin(bufs),
    asio::buffers_begin(bufs) + n);
```

Buffer Debugging

Some standard library implementations, such as the one that ships with Microsoft Visual C++ 8.0 and later, provide a feature called iterator debugging. What this means is that the validity of iterators is checked at runtime. If a program tries to use an iterator that has been invalidated, an assertion will be triggered. For example:

```
std::vector<int> v(1);
std::vector<int>::iterator i = v.begin();
v.clear(); // invalidates iterators
*i = 0; // assertion!
```

Asio takes advantage of this feature to add buffer debugging. Consider the following code:

```
void dont_do_this()
{
    std::string msg = "Hello, world!";
    asio::async_write(sock, asio::buffer(msg), my_handler);
}
```

When you call an asynchronous read or write you need to ensure that the buffers for the operation are valid until the completion handler is called. In the above example, the buffer is the `std::string` variable `msg`. This variable is on the stack, and so it goes out of scope before the asynchronous operation completes. If you're lucky then the application will crash, but random failures are more likely.

When buffer debugging is enabled, Asio stores an iterator into the string until the asynchronous operation completes, and then dereferences it to check its validity. In the above example you would observe an assertion failure just before Asio tries to call the completion handler.

This feature is automatically made available for Microsoft Visual Studio 8.0 or later and for GCC when `_GLIBCXX_DEBUG` is defined. There is a performance cost to this checking, so buffer debugging is only enabled in debug builds. For other compilers it may be enabled by defining `ASIO_ENABLE_BUFFER_DEBUGGING`. It can also be explicitly disabled by defining `ASIO_DISABLE_BUFFER_DEBUGGING`.

See Also

[buffer](#), [buffers_begin](#), [buffers_end](#), [buffers_iterator](#), [const_buffer](#), [const_buffers_1](#), [mutable_buffer](#), [mutable_buffers_1](#), [streambuf](#), [ConstBufferSequence](#), [MutableBufferSequence](#), [buffers example \(C++03\)](#), [buffers example \(c++11\)](#).

1.2.6 Streams, Short Reads and Short Writes

Many I/O objects in Asio are stream-oriented. This means that:

- There are no message boundaries. The data being transferred is a continuous sequence of bytes.
- Read or write operations may transfer fewer bytes than requested. This is referred to as a short read or short write.

Objects that provide stream-oriented I/O model one or more of the following type requirements:

- `SyncReadStream`, where synchronous read operations are performed using a member function called `read_some()`.
- `AsyncReadStream`, where asynchronous read operations are performed using a member function called `async_read_some()`.
- `SyncWriteStream`, where synchronous write operations are performed using a member function called `write_some()`.
- `AsyncWriteStream`, where synchronous write operations are performed using a member function called `async_write_some()`.

Examples of stream-oriented I/O objects include `ip::tcp::socket`, `ssl::stream<>`, `posix::stream_descriptor`, `windows::stream_handle`, etc.

Programs typically want to transfer an exact number of bytes. When a short read or short write occurs the program must restart the operation, and continue to do so until the required number of bytes has been transferred. Asio provides generic functions that do this automatically: `read()`, `async_read()`, `write()` and `async_write()`.

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.

See Also

[async_read\(\)](#), [async_write\(\)](#), [read\(\)](#), [write\(\)](#), [AsyncReadStream](#), [AsyncWriteStream](#), [SyncReadStream](#), [SyncWriteStream](#).

1.2.7 Reactor-Style Operations

Sometimes a program must be integrated with a third-party library that wants to perform the I/O operations itself. To facilitate this, Asio includes a `null_buffers` type that can be used with both read and write operations. A `null_buffers` operation doesn't return until the I/O object is "ready" to perform the operation.

As an example, to perform a non-blocking read something like the following may be used:

```
ip::tcp::socket socket(my_io_service);
...
socket.non_blocking(true);
...
socket.async_read_some(null_buffers(), read_handler);
...
void read_handler(asio::error_code ec)
{
    if (!ec)
    {
        std::vector<char> buf(socket.available());
        socket.read_some(buffer(buf));
    }
}
```

These operations are supported for sockets on all platforms, and for the POSIX stream-oriented descriptor classes.

See Also

[null_buffers](#), [basic_socket::non_blocking\(\)](#), [basic_socket::native_non_blocking\(\)](#), [nonblocking example](#).

1.2.8 Line-Based Operations

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, SMTP and FTP. To more easily permit the implementation of line-based protocols, as well as other protocols that use delimiters, Asio 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()
    {
        asio::async_read_until(socket_, data_, "\r\n",
            boost::bind(&http_connection::handle_request_line, this, _1));
    }

    void handle_request_line(asio::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;
            ...
        }
    }

    ...
}

asio::ip::tcp::socket socket_;
asio::streambuf data_;
};
```

The `streambuf` data member serves as a place to store the data that has been read from the socket before it is searched for the delimiter. It is important to remember that there may be additional data *after* the delimiter. This surplus data should be left in the `streambuf` so that it may be inspected by a subsequent call to `read_until()` or `async_read_until()`.

The delimiters may be specified as a single `char`, a `std::string` or a `boost::regex`. The `read_until()` and `async_read_until()` functions also include overloads that accept a user-defined function object called a match condition. For example, to read data into a `streambuf` until whitespace is encountered:

```
typedef asio::buffers_iterator<
    asio::streambuf::const_buffers_type> iterator;

std::pair<iterator, bool>
match_whitespace(iterator begin, iterator end)
{
    iterator i = begin;
    while (i != end)
        if (std::isspace(*i++))
            return std::make_pair(i, true);
```

```

    return std::make_pair(i, false);
}
...
asio::streambuf b;
asio::read_until(s, b, match whitespace);

```

To read data into a streambuf until a matching character is found:

```

class match_char
{
public:
    explicit match_char(char c) : c_(c) {}

    template <typename Iterator>
    std::pair<Iterator, bool> operator()(Iterator begin, Iterator end) const
    {
        Iterator i = begin;
        while (i != end)
            if (*i == c_)
                return std::make_pair(i, true);
        return std::make_pair(i, false);
    }

private:
    char c_;
};

namespaceasio {
template <> struct is_match_condition<match_char>
    : public boost::true_type {};
} // namespace asio
...
asio::streambuf b;
asio::read_until(s, b, match_char('a'));

```

The `is_match_condition` type trait automatically evaluates to true for functions, and for function objects with a nested `result_type` `typedef`. For other types the trait must be explicitly specialised, as shown above.

See Also

[async_read_until\(\)](#), [is_match_condition](#), [read_until\(\)](#), [streambuf](#), [HTTP client example](#).

1.2.9 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 = asio_handler_allocate(size, &h);
```

Similarly, to deallocate the memory it will execute:

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

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

```
void* asio_handler_allocate(size_t, ...);
void asio_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.

See Also

[asio_handler_allocate](#), [asio_handler_deallocate](#), [custom memory allocation example \(C++03\)](#), [custom memory allocation example \(C++11\)](#).

1.2.10 Handler Tracking

To aid in debugging asynchronous programs, Asio provides support for handler tracking. When enabled by defining `ASIO_ENABLE_HANDLER_TRACKING`, Asio writes debugging output to the standard error stream. The output records asynchronous operations and the relationships between their handlers.

This feature is useful when debugging and you need to know how your asynchronous operations are chained together, or what the pending asynchronous operations are. As an illustration, here is the output when you run the HTTP Server example, handle a single request, then shut down via Ctrl+C:

```
@asio|1298160085.070638|0*1|signal_set@0x7fff50528f40.async_wait
@asio|1298160085.070888|0*2|socket@0x7fff50528f60.async_accept
@asio|1298160085.070913|0|resolver@0x7fff50528e28.cancel
@asio|1298160118.075438|>2|ec=asio.system:0
@asio|1298160118.075472|2*3|socket@0xb39048.async_receive
@asio|1298160118.075507|2*4|socket@0x7fff50528f60.async_accept
@asio|1298160118.075527|<2|
@asio|1298160118.075540|>3|ec=asio.system:0,bytes_transferred=122
@asio|1298160118.075731|3*5|socket@0xb39048.async_send
@asio|1298160118.075778|<3|
@asio|1298160118.075793|>5|ec=asio.system:0,bytes_transferred=156
@asio|1298160118.075831|5|socket@0xb39048.close
@asio|1298160118.075855|<5|
@asio|1298160122.827317|>1|ec=asio.system:0,signal_number=2
@asio|1298160122.827333|1|socket@0x7fff50528f60.close
@asio|1298160122.827359|<1|
@asio|1298160122.827370|>4|ec=asio.system:125
@asio|1298160122.827378|<4|
@asio|1298160122.827394|0|signal_set@0x7fff50528f40.cancel
```

Each line is of the form:

```
<tag>|<timestamp>|<action>|<description>
```

The `<tag>` is always `@asio`, and is used to identify and extract the handler tracking messages from the program output.

The `<timestep>` is seconds and microseconds from 1 Jan 1970 UTC.

The `<action>` takes one of the following forms:

>n The program entered the handler number `n`. The `<description>` shows the arguments to the handler.

<n The program left handler number `n`.

!n The program left handler number `n` due to an exception.

~n The handler number `n` was destroyed without having been invoked. This is usually the case for any unfinished asynchronous operations when the `io_service` is destroyed.

n*m The handler number `n` created a new asynchronous operation with completion handler number `m`. The `<description>` shows what asynchronous operation was started.

n The handler number `n` performed some other operation. The `<description>` shows what function was called. Currently only `close()` and `cancel()` operations are logged, as these may affect the state of pending asynchronous operations.

Where the `<description>` shows a synchronous or asynchronous operation, the format is `<object-type>@<pointer>.` `<operation>`. For handler entry, it shows a comma-separated list of arguments and their values.

As shown above, Each handler is assigned a numeric identifier. Where the handler tracking output shows a handler number of 0, it means that the action was performed outside of any handler.

Visual Representations

The handler tracking output may be post-processed using the included `handlerviz.pl` tool to create a visual representation of the handlers (requires the GraphViz tool `dot`).

1.2.11 Stackless Coroutines

The `coroutine` class provides support for stackless coroutines. Stackless coroutines enable programs to implement asynchronous logic in a synchronous manner, with minimal overhead, as shown in the following example:

```
struct session : asio::coroutine
{
    boost::shared_ptr<tcp::socket> socket_;
    boost::shared_ptr<std::vector<char> > buffer_;

    session(boost::shared_ptr<tcp::socket> socket)
        : socket_(socket),
          buffer_(new std::vector<char>(1024))
    {
    }

    void operator()(asio::error_code ec = asio::error_code(), std::size_t n = 0)
    {
        if (!ec) reenter(this)
        {
            for (;;)
            {
                yield socket_->async_read_some(asio::buffer(*buffer_), *this);
                yield asio::async_write(*socket_, asio::buffer(*buffer_, n), *this);
            }
        }
    }
};
```

The `coroutine` class is used in conjunction with the pseudo-keywords `reenter`, `yield` and `fork`. These are preprocessor macros, and are implemented in terms of a `switch` statement using a technique similar to Duff's Device. The `coroutine` class's documentation provides a complete description of these pseudo-keywords.

See Also

[coroutine](#), [HTTP Server 4 example](#), [Stackful Coroutines](#).

1.2.12 Stackful Coroutines

The [spawn\(\)](#) function is a high-level wrapper for running stackful coroutines. It is based on the Boost.Coroutine library. The `spawn()` function enables programs to implement asynchronous logic in a synchronous manner, as shown in the following example:

```
asio::spawn(my_strand, do_echo);

// ...

void do_echo(asio::yield_context yield)
{
    try
    {
        char data[128];
        for (;;)
        {
            std::size_t length =
                my_socket.async_read_some(
                    asio::buffer(data), yield);

            asio::async_write(my_socket,
                asio::buffer(data, length), yield);
        }
    }
    catch (std::exception& e)
    {
        // ...
    }
}
```

The first argument to `spawn()` may be a [strand](#), [io_service](#), or [completion handler](#). This argument determines the context in which the coroutine is permitted to execute. For example, a server's per-client object may consist of multiple coroutines; they should all run on the same `strand` so that no explicit synchronisation is required.

The second argument is a function object with signature:

```
void coroutine(asio::yield_context yield);
```

that specifies the code to be run as part of the coroutine. The parameter `yield` may be passed to an asynchronous operation in place of the completion handler, as in:

```
std::size_t length =
    my_socket.async_read_some(
        asio::buffer(data), yield);
```

This starts the asynchronous operation and suspends the coroutine. The coroutine will be resumed automatically when the asynchronous operation completes.

Where an asynchronous operation's handler signature has the form:

```
void handler(asio::error_code ec, result_type result);
```

the initiating function returns the `result_type`. In the `async_read_some` example above, this is `size_t`. If the asynchronous operation fails, the `error_code` is converted into a `system_error` exception and thrown.

Where a handler signature has the form:

```
void handler(asio::error_code ec);
```

the initiating function returns `void`. As above, an error is passed back to the coroutine as a `system_error` exception.

To collect the `error_code` from an operation, rather than have it throw an exception, associate the output variable with the `yield_context` as follows:

```
asio::error_code ec;
std::size_t length =
    my_socket.async_read_some(
        asio::buffer(data), yield[ec]);
```

Note: if `spawn()` is used with a custom completion handler of type `Handler`, the function object signature is actually:

```
void coroutine(asio::basic_yield_context<Handler> yield);
```

See Also

[spawn](#), [yield_context](#), [basic_yield_context](#), [Spawn example \(C++03\)](#), [Spawn example \(C++11\)](#), [Stackless Coroutines](#).

1.3 Networking

- [TCP, UDP and ICMP](#)
- [Support for Other Protocols](#)
- [Socket Iostreams](#)
- [The BSD Socket API and Asio](#)

1.3.1 TCP, UDP and ICMP

Asio provides off-the-shelf support for the internet protocols TCP, UDP and ICMP.

TCP Clients

Hostname resolution is performed using a resolver, where host and service names are looked up and converted into one or more endpoints:

```
ip::tcp::resolver resolver(my_io_service);
ip::tcp::resolver::query query("www.boost.org", "http");
ip::tcp::resolver::iterator iter = resolver.resolve(query);
ip::tcp::resolver::iterator end; // End marker.
while (iter != end)
{
    ip::tcp::endpoint endpoint = *iter++;
    std::cout << endpoint << std::endl;
}
```

The list of endpoints obtained above could contain both IPv4 and IPv6 endpoints, so a program should try each of them until it finds one that works. This keeps the client program independent of a specific IP version.

To simplify the development of protocol-independent programs, TCP clients may establish connections using the free functions `connect()` and `async_connect()`. These operations try each endpoint in a list until the socket is successfully connected. For example, a single call:

```
ip::tcp::socket socket(my_io_service);
asio::connect(socket, resolver.resolve(query));
```

will synchronously try all endpoints until one is successfully connected. Similarly, an asynchronous connect may be performed by writing:

```
asio::async_connect(socket_, iter,
    boost::bind(&client::handle_connect, this,
        asio::placeholders::error));

// ...

void handle_connect(const error_code& error)
{
    if (!error)
    {
        // Start read or write operations.
    }
    else
    {
        // Handle error.
    }
}
```

When a specific endpoint is available, a socket can be created and connected:

```
ip::tcp::socket socket(my_io_service);
socket.connect(endpoint);
```

Data may be read from or written to a connected TCP socket using the `receive()`, `async_receive()`, `send()` or `async_send()` member functions. However, as these could result in [short writes or reads](#), an application will typically use the following operations instead: `read()`, `async_read()`, `write()` and `async_write()`.

TCP Servers

A program uses an acceptor to accept incoming TCP connections:

```
ip::tcp::acceptor acceptor(my_io_service, my_endpoint);
...
ip::tcp::socket socket(my_io_service);
acceptor.accept(socket);
```

After a socket has been successfully accepted, it may be read from or written to as illustrated for TCP clients above.

UDP

UDP hostname resolution is also performed using a resolver:

```
ip::udp::resolver resolver(my_io_service);
ip::udp::resolver::query query("localhost", "daytime");
ip::udp::resolver::iterator iter = resolver.resolve(query);
...
```

A UDP socket is typically bound to a local endpoint. The following code will create an IP version 4 UDP socket and bind it to the "any" address on port 12345:

```
ip::udp::endpoint endpoint(ip::udp::v4(), 12345);
ip::udp::socket socket(my_io_service, endpoint);
```

Data may be read from or written to an unconnected UDP socket using the `receive_from()`, `async_receive_from()`, `send_to()` or `async_send_to()` member functions. For a connected UDP socket, use the `receive()`, `async_receive()`, `send()` or `async_send()` member functions.

ICMP

As with TCP and UDP, ICMP hostname resolution is performed using a resolver:

```
ip::icmp::resolver resolver(my_io_service);
ip::icmp::resolver::query query("localhost", "");
ip::icmp::resolver::iterator iter = resolver.resolve(query);
...
```

An ICMP socket may be bound to a local endpoint. The following code will create an IP version 6 ICMP socket and bind it to the "any" address:

```
ip::icmp::endpoint endpoint(ip::icmp::v6(), 0);
ip::icmp::socket socket(my_io_service, endpoint);
```

The port number is not used for ICMP.

Data may be read from or written to an unconnected ICMP socket using the [receive_from\(\)](#), [async_receive_from\(\)](#), [send_to\(\)](#) or [async_send_to\(\)](#) member functions.

See Also

[ip::tcp](#), [ip::udp](#), [ip::icmp](#), [daytime protocol tutorials](#), [ICMP ping example](#).

1.3.2 Support for Other Protocols

Support for other socket protocols (such as Bluetooth or IRCOMM sockets) can be added by implementing the [protocol type requirements](#). However, in many cases these protocols may also be used with Asio's generic protocol support. For this, Asio provides the following four classes:

- [generic::datagram_protocol](#)
- [generic::raw_protocol](#)
- [generic::seq_packet_protocol](#)
- [generic::stream_protocol](#)

These classes implement the [protocol type requirements](#), but allow the user to specify the address family (e.g. `AF_INET`) and protocol type (e.g. `IPPROTO_TCP`) at runtime. For example:

```
asio::generic::stream_protocol::socket my_socket(my_io_service);
my_socket.open(asio::generic::stream_protocol(AF_INET, IPPROTO_TCP));
...
```

An endpoint class template, [asio::generic::basic_endpoint](#), is included to support these protocol classes. This endpoint can hold any other endpoint type, provided its native representation fits into a `sockaddr_storage` object. This class will also convert from other types that implement the [endpoint](#) type requirements:

```
asio::ip::tcp::endpoint my_endpoint1 = ...;
asio::generic::stream_protocol::endpoint my_endpoint2(my_endpoint1);
```

The conversion is implicit, so as to support the following use cases:

```
asio::generic::stream_protocol::socket my_socket(my_io_service);
asio::ip::tcp::endpoint my_endpoint = ...;
my_socket.connect(my_endpoint);
```

C++11 Move Construction

When using C++11, it is possible to perform move construction from a socket (or acceptor) object to convert to the more generic protocol's socket (or acceptor) type. If the protocol conversion is valid:

```
Protocol1 p1 = ...;
Protocol2 p2(p1);
```

then the corresponding socket conversion is allowed:

```
Protocol1::socket my_socket1(my_io_service);
...
Protocol2::socket my_socket2(std::move(my_socket1));
```

For example, one possible conversion is from a TCP socket to a generic stream-oriented socket:

```
asio::ip::tcp::socket my_socket1(my_io_service);
...
asio::generic::stream_protocol::socket my_socket2(std::move(my_socket1));
```

These conversions are also available for move-assignment.

These conversions are not limited to the above generic protocol classes. User-defined protocols may take advantage of this feature by similarly ensuring the conversion from `Protocol1` to `Protocol2` is valid, as above.

Accepting Generic Sockets

As a convenience, a socket acceptor's `accept()` and `async_accept()` functions can directly accept into a different protocol's socket type, provided the corresponding protocol conversion is valid. For example, the following is supported because the protocol `asio::ip::tcp` is convertible to `asio::generic::stream_protocol`:

```
asio::ip::tcp::acceptor my_acceptor(my_io_service);
...
asio::generic::stream_protocol::socket my_socket(my_io_service);
my_acceptor.accept(my_socket);
```

See Also

[generic::datagram_protocol](#), [generic::raw_protocol](#), [generic::seq_packet_protocol](#), [generic::stream_protocol](#), [protocol type requirements](#).

1.3.3 Socket Iostreams

Asio includes classes that implement iostreams on top of sockets. These hide away the complexities associated with endpoint resolution, protocol independence, etc. 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 acceptor(ios, endpoint);

for (;;)
{
    ip::tcp::iostream stream;
    acceptor.accept(*stream.rdbuf());
    ...
}

```

Timeouts may be set by calling `expires_at()` or `expires_from_now()` to establish a deadline. Any socket operations that occur past the deadline will put the iostream into a "bad" state.

For example, a simple client program like this:

```

ip::tcp::iostream stream;
stream.expires_from_now(boost::posix_time::seconds(60));
stream.connect("www.boost.org", "http");
stream << "GET /LICENSE_1_0.txt HTTP/1.0\r\n";
stream << "Host: www.boost.org\r\n";
stream << "Accept: */*\r\n";
stream << "Connection: close\r\n\r\n";
stream.flush();
std::cout << stream.rdbuf();

```

will fail if all the socket operations combined take longer than 60 seconds.

If an error does occur, the iostream's `error()` member function may be used to retrieve the error code from the most recent system call:

```

if (!stream)
{
    std::cout << "Error: " << stream.error().message() << "\n";
}

```

See Also

[ip::tcp::iostream](#), [basic_socket_iostream](#), [iostreams examples](#).

Notes

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

1.3.4 The BSD Socket API and Asio

The Asio library includes a low-level socket interface based on the BSD socket 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 socket API not included. For example, the use of `int` to represent all sockets lacks type safety. The socket representation in Asio uses a distinct type for each protocol, e.g. for TCP one would use `ip::tcp::socket`, and for UDP one uses `ip::udp::socket`.

The following table shows the mapping between the BSD socket API and Asio:

BSD Socket API Elements	Equivalents in Asio
socket descriptor - int (POSIX) or SOCKET (Windows)	For TCP: <code>ip::tcp::socket</code> , <code>ip::tcp::acceptor</code> For UDP: <code>ip::udp::socket</code> <code>basic_socket</code> , <code>basic_stream_socket</code> , <code>basic_datagram_socket</code> , <code>basic_raw_socket</code>
<code>in_addr</code> , <code>in6_addr</code>	<code>ip::address</code> , <code>ip::address_v4</code> , <code>ip::address_v6</code>
<code>sockaddr_in</code> , <code>sockaddr_in6</code>	For TCP: <code>ip::tcp::endpoint</code> For UDP: <code>ip::udp::endpoint</code> <code>ip::basic_endpoint</code>
<code>accept()</code>	For TCP: <code>ip::tcp::acceptor::accept()</code> <code>basic_socket_acceptor::accept()</code>
<code>bind()</code>	For TCP: <code>ip::tcp::acceptor::bind()</code> , <code>ip::tcp::socket::bind()</code> For UDP: <code>ip::udp::socket::bind()</code> <code>basic_socket::bind()</code>
<code>close()</code>	For TCP: <code>ip::tcp::acceptor::close()</code> , <code>ip::tcp::socket::close()</code> For UDP: <code>ip::udp::socket::close()</code> <code>basic_socket::close()</code>
<code>connect()</code>	For TCP: <code>ip::tcp::socket::connect()</code> For UDP: <code>ip::udp::socket::connect()</code> <code>basic_socket::connect()</code>
<code>getaddrinfo()</code> , <code>gethostbyaddr()</code> , <code>gethostbyname()</code> , <code>getnameinfo()</code> , <code>getservbyname()</code> , <code>getservbyport()</code>	For TCP: <code>ip::tcp::resolver::resolve()</code> , <code>ip::tcp::resolver::async_resolve()</code> For UDP: <code>ip::udp::resolver::resolve()</code> , <code>ip::udp::resolver::async_resolve()</code> <code>ip::basic_resolver::resolve()</code> , <code>ip::basic_resolver::async_resolve()</code>
<code>gethostname()</code>	<code>ip::host_name()</code>
<code>getpeername()</code>	For TCP: <code>ip::tcp::socket::remote_endpoint()</code> For UDP: <code>ip::udp::socket::remote_endpoint()</code> <code>basic_socket::remote_endpoint()</code>
<code>getsockname()</code>	For TCP: <code>ip::tcp::acceptor::local_endpoint()</code> , <code>ip::tcp::socket::local_endpoint()</code> For UDP: <code>ip::udp::socket::local_endpoint()</code> <code>basic_socket::local_endpoint()</code>
<code>getsockopt()</code>	For TCP: <code>ip::tcp::acceptor::get_option()</code> , <code>ip::tcp::socket::get_option()</code> For UDP: <code>ip::udp::socket::get_option()</code> <code>basic_socket::get_option()</code>
<code>inet_addr()</code> , <code>inet_aton()</code> , <code>inet_ntop()</code>	<code>ip::address::from_string()</code> , <code>ip::address_v4::from_string()</code> , <code>ip_address_v6::from_string()</code>
<code>inet_ntoa()</code> , <code>inet_ntop()</code>	<code>ip::address::to_string()</code> , <code>ip::address_v4::to_string()</code> , <code>ip_address_v6::to_string()</code>
<code>ioctl()</code>	For TCP: <code>ip::tcp::socket::io_control()</code> For UDP: <code>ip::udp::socket::io_control()</code> <code>basic_socket::io_control()</code>
<code>listen()</code>	For TCP: <code>ip::tcp::acceptor::listen()</code> <code>basic_socket_acceptor::listen()</code>
<code>poll()</code> , <code>select()</code> , <code>pselect()</code>	<code>io_service::run()</code> , <code>io_service::run_one()</code> , <code>io_service::poll()</code> , <code>io_service::poll_one()</code> Note: in conjunction with asynchronous operations.

BSD Socket API Elements	Equivalents in Asio
readv(), recv(), read()	For TCP: ip::tcp::socket::read_some(), ip::tcp::socket::async_read_some(), ip::tcp::socket::receive(), ip::tcp::socket::async_receive() For UDP: ip::udp::socket::receive(), ip::udp::socket::async_receive() basic_stream_socket::read_some(), basic_stream_socket::async_read_some(), basic_stream_socket::receive(), basic_stream_socket::async_receive(), basic_datagram_socket::receive(), basic_datagram_socket::async_receive()
recvfrom()	For UDP: ip::udp::socket::receive_from(), ip::udp::socket::async_receive_from() basic_datagram_socket::receive_from(), basic_datagram_socket::async_receive_from()
send(), write(), writev()	For TCP: ip::tcp::socket::write_some(), ip::tcp::socket::async_write_some(), ip::tcp::socket::send(), ip::tcp::socket::async_send() For UDP: ip::udp::socket::send(), ip::udp::socket::async_send() basic_stream_socket::write_some(), basic_stream_socket::async_write_some(), basic_stream_socket::send(), basic_stream_socket::async_send(), basic_datagram_socket::send(), basic_datagram_socket::async_send()
sendto()	For UDP: ip::udp::socket::send_to(), ip::udp::socket::async_send_to() basic_datagram_socket::send_to(), basic_datagram_socket::async_send_to()
setsockopt()	For TCP: ip::tcp::acceptor::set_option(), ip::tcp::socket::set_option() For UDP: ip::udp::socket::set_option() basic_socket::set_option()
shutdown()	For TCP: ip::tcp::socket::shutdown() For UDP: ip::udp::socket::shutdown() basic_socket::shutdown()
sockatmark()	For TCP: ip::tcp::socket::at_mark() basic_socket::at_mark()
socket()	For TCP: ip::tcp::acceptor::open(), ip::tcp::socket::open() For UDP: ip::udp::socket::open() basic_socket::open()
socketpair()	local::connect_pair() Note: POSIX operating systems only.

1.4 Timers

Long running I/O operations will often have a deadline by which they must have completed. These deadlines may be expressed as absolute times, but are often calculated relative to the current time.

As a simple example, to perform a synchronous wait operation on a timer using a relative time one may write:

```
io_service i;
...
```

```
deadline_timer t(i);
t.expires_from_now(boost::posix_time::seconds(5));
t.wait();
```

More commonly, a program will perform an asynchronous wait operation on a timer:

```
void handler(asio::error_code ec) { ... }

...
io_service i;
...
deadline_timer t(i);
t.expires_from_now(boost::posix_time::milliseconds(400));
t.async_wait(handler);
...
i.run();
```

The deadline associated with a timer may also be obtained as a relative time:

```
boost::posix_time::time_duration time_until_expiry
= t.expires_from_now();
```

or as an absolute time to allow composition of timers:

```
deadline_timer t2(i);
t2.expires_at(t.expires_at() + boost::posix_time::seconds(30));
```

See Also

[basic_deadline_timer](#), [deadline_timer](#), [deadline_timer_service](#), [timer tutorials](#).

1.5 Serial Ports

Asio includes classes for creating and manipulating serial ports in a portable manner. For example, a serial port may be opened using:

```
serial_port port(my_io_service, name);
```

where name is something like "COM1" on Windows, and "/dev/ttys0" on POSIX platforms.

Once opened, the serial port may be used as a [stream](#). This means the objects can be used with any of the [read\(\)](#), [async_read\(\)](#), [write\(\)](#), [async_write\(\)](#), [read_until\(\)](#) or [async_read_until\(\)](#) free functions.

The serial port implementation also includes option classes for configuring the port's baud rate, flow control type, parity, stop bits and character size.

See Also

[serial_port](#), [serial_port_base](#), [basic_serial_port](#), [serial_port_service](#), [serial_port_base::baud_rate](#), [serial_port_base::flow_control](#), [serial_port_base::parity](#), [serial_port_base::stop_bits](#), [serial_port_base::character_size](#).

Notes

Serial ports are available on all POSIX platforms. For Windows, serial ports are only available at compile time when the I/O completion port backend is used (which is the default). A program may test for the macro ASIO_HAS_SERIAL_PORT to determine whether they are supported.

1.6 Signal Handling

Asio supports signal handling using a class called [signal_set](#). Programs may add one or more signals to the set, and then perform an `async_wait()` operation. The specified handler will be called when one of the signals occurs. The same signal number may be registered with multiple [signal_set](#) objects, however the signal number must be used only with Asio.

```
void handler(
    const asio::error_code& error,
    int signal_number)
{
    if (!error)
    {
        // A signal occurred.
    }
}

...

// Construct a signal set registered for process termination.
asio::signal_set signals(io_service, SIGINT, SIGTERM);

// Start an asynchronous wait for one of the signals to occur.
signals.async_wait(handler);
```

Signal handling also works on Windows, as the Microsoft Visual C++ runtime library maps console events like Ctrl+C to the equivalent signal.

See Also

[signal_set](#), [HTTP server example \(C++03\)](#), [HTTP server example \(C++11\)](#).

1.7 POSIX-Specific Functionality

[UNIX Domain Sockets](#)

[Stream-Oriented File Descriptors](#)

[Fork](#)

1.7.1 UNIX Domain Sockets

Asio provides basic support for UNIX domain sockets (also known as local sockets). The simplest use involves creating a pair of connected sockets. The following code:

```
local::stream_protocol::socket socket1(my_io_service);
local::stream_protocol::socket socket2(my_io_service);
local::connect_pair(socket1, socket2);
```

will create a pair of stream-oriented sockets. To do the same for datagram-oriented sockets, use:

```
local::datagram_protocol::socket socket1(my_io_service);
local::datagram_protocol::socket socket2(my_io_service);
local::connect_pair(socket1, socket2);
```

A UNIX domain socket server may be created by binding an acceptor to an endpoint, in much the same way as one does for a TCP server:

```
::unlink("/tmp/foobar"); // Remove previous binding.  
local::stream_protocol::endpoint ep("/tmp/foobar");  
local::stream_protocol::acceptor acceptor(my_io_service, ep);  
local::stream_protocol::socket socket(my_io_service);  
acceptor.accept(socket);
```

A client that connects to this server might look like:

```
local::stream_protocol::endpoint ep("/tmp/foobar");  
local::stream_protocol::socket socket(my_io_service);  
socket.connect(ep);
```

Transmission of file descriptors or credentials across UNIX domain sockets is not directly supported within Asio, but may be achieved by accessing the socket's underlying descriptor using the [native_handle\(\)](#) member function.

See Also

[local::connect_pair](#), [local::datagram_protocol](#), [local::datagram_protocol::endpoint](#), [local::datagram_protocol::socket](#), [local::stream_protocol](#), [local::stream_protocol::acceptor](#), [local::stream_protocol::endpoint](#), [local::stream_protocol::iostream](#), [local::stream_protocol::socket](#), [UNIX domain sockets examples](#).

Notes

UNIX domain sockets are only available at compile time if supported by the target operating system. A program may test for the macro `ASIO_HAS_LOCAL_SOCKETS` to determine whether they are supported.

1.7.2 Stream-Oriented File Descriptors

Asio includes classes added to permit synchronous and asynchronous read and write operations to be performed on POSIX file descriptors, such as pipes, standard input and output, and various devices (but *not* regular files).

For example, to perform read and write operations on standard input and output, the following objects may be created:

```
posix::stream_descriptor in(my_io_service, ::dup(STDIN_FILENO));  
posix::stream_descriptor out(my_io_service, ::dup(STDOUT_FILENO));
```

These are then used as synchronous or asynchronous read and write streams. This means the objects can be used with any of the [read\(\)](#), [async_read\(\)](#), [write\(\)](#), [async_write\(\)](#), [read_until\(\)](#) or [async_read_until\(\)](#) free functions.

See Also

[posix::stream_descriptor](#), [posix::basic_stream_descriptor](#), [posix::stream_descriptor_service](#), [Chat example \(C++03\)](#), [Chat example \(C++11\)](#).

Notes

POSIX stream descriptors are only available at compile time if supported by the target operating system. A program may test for the macro `ASIO_HAS_POSIX_STREAM_DESCRIPTOR` to determine whether they are supported.

1.7.3 Fork

Asio supports programs that utilise the `fork()` system call. Provided the program calls `io_service.notify_fork()` at the appropriate times, Asio will recreate any internal file descriptors (such as the "self-pipe trick" descriptor used for waking up a reactor). The notification is usually performed as follows:

```
io_service_.notify_fork(asio::io_service::fork_prepare);
if (fork() == 0)
{
    io_service_.notify_fork(asio::io_service::fork_child);
    ...
}
else
{
    io_service_.notify_fork(asio::io_service::fork_parent);
    ...
}
```

User-defined services can also be made fork-aware by overriding the `io_service::service::fork_service()` virtual function.

Note that any file descriptors accessible via Asio's public API (e.g. the descriptors underlying `basic_socket<>`, `posix::stream_descriptor`, etc.) are not altered during a fork. It is the program's responsibility to manage these as required.

See Also

[io_service::notify_fork\(\)](#), [io_service::fork_event](#), [io_service::service::fork_service\(\)](#), [Fork examples](#).

1.8 Windows-Specific Functionality

[Stream-Oriented HANDLES](#)

[Random-Access HANDLES](#)

[Object HANDLES](#)

1.8.1 Stream-Oriented HANDLES

Asio contains classes to allow asynchronous read and write operations to be performed on Windows HANDLES, such as named pipes.

For example, to perform asynchronous operations on a named pipe, the following object may be created:

```
HANDLE handle = ::CreateFile(...);
windows::stream_handle pipe(my_io_service, handle);
```

These are then used as synchronous or asynchronous read and write streams. This means the objects can be used with any of the `read()`, `async_read()`, `write()`, `async_write()`, `read_until()` or `async_read_until()` free functions.

The kernel object referred to by the HANDLE must support use with I/O completion ports (which means that named pipes are supported, but anonymous pipes and console streams are not).

See Also

[windows::stream_handle](#), [windows::basic_stream_handle](#), [windows::stream_handle_service](#).

Notes

Windows stream HANDLES are only available at compile time when targeting Windows and only when the I/O completion port backend is used (which is the default). A program may test for the macro `ASIO_HAS_WINDOWS_STREAM_HANDLE` to determine whether they are supported.

1.8.2 Random-Access HANDLES

Asio provides Windows-specific classes that permit asynchronous read and write operations to be performed on HANDLES that refer to regular files.

For example, to perform asynchronous operations on a file the following object may be created:

```
HANDLE handle = ::CreateFile(...);
windows::random_access_handle file(my_io_service, handle);
```

Data may be read from or written to the handle using one of the `read_some_at()`, `async_read_some_at()`, `write_some_at()` or `async_write_some_at()` member functions. However, like the equivalent functions (`read_some()`, etc.) on streams, these functions are only required to transfer one or more bytes in a single operation. Therefore free functions called `read_at()`, `async_read_at()`, `write_at()` and `async_write_at()` have been created to repeatedly call the corresponding `*_some_at()` function until all data has been transferred.

See Also

[windows::random_access_handle](#), [windows::basic_random_access_handle](#), [windows::random_access_handle_service](#).

Notes

Windows random-access HANDLES are only available at compile time when targeting Windows and only when the I/O completion port backend is used (which is the default). A program may test for the macro `ASIO_HAS_WINDOWS_RANDOM_ACCESS_HANDLE` to determine whether they are supported.

1.8.3 Object HANDLES

Asio provides Windows-specific classes that permit asynchronous wait operations to be performed on HANDLES to kernel objects of the following types:

- Change notification
- Console input
- Event
- Memory resource notification
- Process
- Semaphore
- Thread
- Waitable timer

For example, to perform asynchronous operations on an event, the following object may be created:

```
HANDLE handle = ::CreateEvent(...);
windows::object_handle file(my_io_service, handle);
```

The `wait()` and `async_wait()` member functions may then be used to wait until the kernel object is signalled.

See Also

[windows::object_handle](#), [windows::basic_object_handle](#), [windows::object_handle_service](#).

Notes

Windows object `HANDLE`s are only available at compile time when targeting Windows. Programs may test for the macro `ASIO_HANDLE` or `S_WINDOWS_OBJECT_HANDLE` to determine whether they are supported.

1.9 SSL

Asio contains classes and class templates for basic SSL support. These classes allow encrypted communication to be layered on top of an existing stream, such as a TCP socket.

Before creating an encrypted stream, an application must construct an SSL context object. This object is used to set SSL options such as verification mode, certificate files, and so on. As an illustration, client-side initialisation may look something like:

```
ssl::context ctx(ssl::context::sslv23);
ctx.set_verify_mode(ssl::verify_peer);
ctx.load_verify_file("ca.pem");
```

To use SSL with a TCP socket, one may write:

```
ssl::stream<ip::tcp::socket> ssl_sock(my_io_service, ctx);
```

To perform socket-specific operations, such as establishing an outbound connection or accepting an incoming one, the underlying socket must first be obtained using the `ssl::stream` template's `lowest_layer()` member function:

```
ip::tcp::socket::lowest_layer_type& sock = ssl_sock.lowest_layer();
sock.connect(my_endpoint);
```

In some use cases the underlying stream object will need to have a longer lifetime than the SSL stream, in which case the template parameter should be a reference to the stream type:

```
ip::tcp::socket sock(my_io_service);
ssl::stream<ip::tcp::socket&> ssl_sock(sock, ctx);
```

SSL handshaking must be performed prior to transmitting or receiving data over an encrypted connection. This is accomplished using the `ssl::stream` template's `handshake()` or `async_handshake()` member functions.

Once connected, SSL stream objects are used as synchronous or asynchronous read and write streams. This means the objects can be used with any of the `read()`, `async_read()`, `write()`, `async_write()`, `read_until()` or `async_read_until()` free functions.

Certificate Verification

Asio provides various methods for configuring the way SSL certificates are verified:

- `ssl::context::set_default_verify_paths()`
- `ssl::context::set_verify_mode()`
- `ssl::context::set_verify_callback()`
- `ssl::context::load_verify_file()`
- `ssl::stream::set_verify_mode()`
- `ssl::stream::set_verify_callback()`

To simplify use cases where certificates are verified according to the rules in RFC 2818 (certificate verification for HTTPS), Asio provides a reusable verification callback as a function object:

- `ssl::rfc2818_verification`

The following example shows verification of a remote host's certificate according to the rules used by HTTPS:

```
using asio::ip::tcp;
namespace ssl = asio::ssl;
typedef ssl::stream<tcp::socket> ssl_socket;

// Create a context that uses the default paths for
// finding CA certificates.
ssl::context ctx(ssl::context::sslv23);
ctx.set_default_verify_paths();

// Open a socket and connect it to the remote host.
asio::io_service io_service;
ssl_socket sock(io_service, ctx);
tcp::resolver resolver(io_service);
tcp::resolver::query query("host.name", "https");
asio::connect(sock.lowest_layer(), resolver.resolve(query));
sock.lowest_layer().set_option(tcp::no_delay(true));

// Perform SSL handshake and verify the remote host's
// certificate.
sock.set_verify_mode(ssl::verify_peer);
sock.set_verify_callback(ssl::rfc2818_verification("host.name"));
sock.handshake(ssl_socket::client);

// ... read and write as normal ...
```

SSL and Threads

SSL stream objects perform no locking of their own. Therefore, it is essential that all asynchronous SSL operations are performed in an implicit or explicit [strand](#). Note that this means that no synchronisation is required (and so no locking overhead is incurred) in single threaded programs.

See Also

[ssl::context](#), [ssl::rfc2818_verification](#), [ssl::stream](#), [SSL example](#).

Notes

[OpenSSL](#) is required to make use of Asio's SSL support. When an application needs to use OpenSSL functionality that is not wrapped by Asio, the underlying OpenSSL types may be obtained by calling [ssl::context::native_handle\(\)](#) or [ssl::stream::native_handle\(\)](#).

1.10 C++ 2011 Support

[System Errors and Error Codes](#)

[Movable I/O Objects](#)

[Movable Handlers](#)

[Variadic Templates](#)

[Array Container](#)

[Atomics](#)

[Shared Pointers](#)

[Chrono](#)

[Futures](#)

1.10.1 System Errors and Error Codes

When available, Asio can use the `std::error_code` and `std::system_error` classes for reporting errors. In this case, the names `asio::error_code` and `asio::system_error` will be typecasts for these standard classes.

System error support is automatically enabled for g++ 4.6 and later, when the `-std=c++0x` or `-std=gnu++0x` compiler options are used. It may be disabled by defining `ASIO_DISABLE_STD_SYSTEM_ERROR`, or explicitly enabled for other compilers by defining `ASIO_HAS_STD_SYSTEM_ERROR`.

1.10.2 Movable I/O Objects

When move support is available (via rvalue references), Asio allows move construction and assignment of sockets, serial ports, POSIX descriptors and Windows handles.

Move support allows you to write code like:

```
tcp::socket make_socket(io_service& i)
{
    tcp::socket s(i);
    ...
    std::move(s);
}
```

or:

```
class connection : public enable_shared_from_this<connection>
{
private:
    tcp::socket socket_;
    ...
public:
    connection(tcp::socket&& s) : socket_(std::move(s)) {}
    ...
};

class server
{
private:
    tcp::acceptor acceptor_;
    tcp::socket socket_;
    ...
    void handle_accept(error_code ec)
    {
        if (!ec)
            std::make_shared<connection>(std::move(socket_))->go();
        acceptor_.async_accept(socket_, ...);
    }
    ...
};
```

as well as:

```
std::vector<tcp::socket> sockets;
sockets.push_back(tcp::socket(...));
```

A word of warning: There is nothing stopping you from moving these objects while there are pending asynchronous operations, but it is unlikely to be a good idea to do so. In particular, composed operations like `async_read()` store a reference to the stream object. Moving during the composed operation means that the composed operation may attempt to access a moved-from object.

Move support is automatically enabled for g++ 4.5 and later, when the `-std=c++0x` or `-std=gnu++0x` compiler options are used. It may be disabled by defining `ASIO_DISABLE_MOVE`, or explicitly enabled for other compilers by defining `ASIO_HAS_MOVE`. Note that these macros also affect the availability of [movable handlers](#).

1.10.3 Movable Handlers

As an optimisation, user-defined completion handlers may provide move constructors, and Asio's implementation will use a handler's move constructor in preference to its copy constructor. In certain circumstances, Asio may be able to eliminate all calls to a handler's copy constructor. However, handler types are still required to be copy constructible.

When move support is enabled, asynchronous that are documented as follows:

```
template <typename Handler>
void async_XYZ(..., Handler handler);
```

are actually declared as:

```
template <typename Handler>
void async_XYZ(..., Handler&& handler);
```

The handler argument is perfectly forwarded and the move construction occurs within the body of `async_XYZ()`. This ensures that all other function arguments are evaluated prior to the move. This is critical when the other arguments to `async_XYZ()` are members of the handler. For example:

```
struct my_operation
{
    shared_ptr<tcp::socket> socket;
    shared_ptr<vector<char>> buffer;
    ...
    void operator(error_code ec, size_t length)
    {
        ...
        socket->async_read_some(asio::buffer(*buffer), std::move(*this));
        ...
    }
};
```

Move support is automatically enabled for g++ 4.5 and later, when the `-std=c++0x` or `-std=gnu++0x` compiler options are used. It may be disabled by defining `ASIO_DISABLE_MOVE`, or explicitly enabled for other compilers by defining `ASIO_HAS_MOVE`. Note that these macros also affect the availability of [movable I/O objects](#).

1.10.4 Variadic Templates

When supported by a compiler, Asio can use variadic templates to implement the `basic_socket_streambuf::connect()` and `basic_socket_iostream::functions`.

Support for variadic templates is automatically enabled for g++ 4.3 and later, when the `-std=c++0x` or `-std=gnu++0x` compiler options are used. It may be disabled by defining `ASIO_DISABLE_VARIADIC_TEMPLATES`, or explicitly enabled for other compilers by defining `ASIO_HAS_VARIADIC_TEMPLATES`.

1.10.5 Array Container

Where the standard library provides `std::array<>`, Asio:

- Provides overloads for the `buffer()` function.
- Uses it in preference to `boost::array<>` for the `ip::address_v4::bytes_type` and `ip::address_v6::bytes_type` types.

- Uses it in preference to `boost::array`<> where a fixed size array type is needed in the implementation.

Support for `std::array`<> is automatically enabled for g++ 4.3 and later, when the `-std=c++0x` or `-std=gnu++0x` compiler options are used, as well as for Microsoft Visual C++ 10. It may be disabled by defining `ASIO_DISABLE_STD_ARRAY`, or explicitly enabled for other compilers by defining `ASIO_HAS_STD_ARRAY`.

1.10.6 Atomics

Asio's implementation can use `std::atomic`<> in preference to `boost::detail::atomic_count`.

Support for the standard atomic integer template is automatically enabled for g++ 4.5 and later, when the `-std=c++0x` or `-std=gnu++0x` compiler options are used. It may be disabled by defining `ASIO_DISABLE_STD_ATOMIC`, or explicitly enabled for other compilers by defining `ASIO_HAS_STD_ATOMIC`.

1.10.7 Shared Pointers

Asio's implementation can use `std::shared_ptr`<> and `std::weak_ptr`<> in preference to the Boost equivalents.

Support for the standard smart pointers is automatically enabled for g++ 4.3 and later, when the `-std=c++0x` or `-std=gnu++0x` compiler options are used, as well as for Microsoft Visual C++ 10. It may be disabled by defining `ASIO_DISABLE_STD_SHARED_PTR`, or explicitly enabled for other compilers by defining `ASIO_HAS_STD_SHARED_PTR`.

1.10.8 Chrono

Asio provides timers based on the `std::chrono` facilities via the `basic_waitable_timer` class template. The typedefs `system_timer`, `steady_timer` and `high_resolution_timer` utilise the standard clocks `system_clock`, `steady_clock` and `high_resolution_clock` respectively.

Support for the `std::chrono` facilities is automatically enabled for g++ 4.6 and later, when the `-std=c++0x` or `-std=gnu+0x` compiler options are used. (Note that, for g++, the draft-standard `monotonic_clock` is used in place of `steady_clock`.) Support may be disabled by defining `ASIO_DISABLE_STD_CHRONO`, or explicitly enabled for other compilers by defining `ASIO_HAS_STD_CHRONO`.

When `standard chrono` is unavailable, Asio will otherwise use the Boost.Chrono library. The `basic_waitable_timer` class template may be used with either.

1.10.9 Futures

The `asio::use_future` special value provides first-class support for returning a C++11 `std::future` from an asynchronous operation's initiating function.

To use `asio::use_future`, pass it to an asynchronous operation instead of a normal completion handler. For example:

```
std::future<std::size_t> length =
    my_socket.async_read_some(my_buffer, asio::use_future);
```

Where a handler signature has the form:

```
void handler(asio::error_code ec, result_type result);
```

the initiating function returns a `std::future` templated on `result_type`. In the above example, this is `std::size_t`. If the asynchronous operation fails, the `error_code` is converted into a `system_error` exception and passed back to the caller through the future.

Where a handler signature has the form:

```
void handler(asio::error_code ec);
```

the initiating function returns `std::future<void>`. As above, an error is passed back in the future as a `system_error` exception.

[use_future, use_future_t, Futures example \(C++11\)](#).

1.11 Platform-Specific Implementation Notes

This section lists platform-specific implementation details, such as the default demultiplexing mechanism, the number of threads created internally, and when threads are created.

Linux Kernel 2.4

Demultiplexing mechanism:

- Uses `select` for demultiplexing. This means that the number of file descriptors in the process cannot be permitted to exceed `FD_SETSIZE`.

Threads:

- Demultiplexing using `select` is performed in one of the threads that calls `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- At most `min(64, IOV_MAX)` buffers may be transferred in a single operation.

Linux Kernel 2.6

Demultiplexing mechanism:

- Uses `epoll` for demultiplexing.

Threads:

- Demultiplexing using `epoll` is performed in one of the threads that calls `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- At most `min(64, IOV_MAX)` buffers may be transferred in a single operation.

Solaris

Demultiplexing mechanism:

- Uses `/dev/poll` for demultiplexing.

Threads:

- Demultiplexing using `/dev/poll` is performed in one of the threads that calls `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- At most `min(64, IOV_MAX)` buffers may be transferred in a single operation.

QNX Neutrino

Demultiplexing mechanism:

- Uses `select` for demultiplexing. This means that the number of file descriptors in the process cannot be permitted to exceed `FD_SETSIZE`.

Threads:

- Demultiplexing using `select` is performed in one of the threads that calls `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- At most `min(64, IOV_MAX)` buffers may be transferred in a single operation.

Mac OS X

Demultiplexing mechanism:

- Uses `kqueue` for demultiplexing.

Threads:

- Demultiplexing using `kqueue` is performed in one of the threads that calls `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- At most `min(64, IOV_MAX)` buffers may be transferred in a single operation.

FreeBSD

Demultiplexing mechanism:

- Uses `kqueue` for demultiplexing.

Threads:

- Demultiplexing using `kqueue` is performed in one of the threads that calls `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- At most `min(64, IOV_MAX)` buffers may be transferred in a single operation.

AIX

Demultiplexing mechanism:

- Uses `select` for demultiplexing. This means that the number of file descriptors in the process cannot be permitted to exceed `FD_SETSIZE`.

Threads:

- Demultiplexing using `select` is performed in one of the threads that calls `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- At most `min(64, IOV_MAX)` buffers may be transferred in a single operation.

HP-UX

Demultiplexing mechanism:

- Uses `select` for demultiplexing. This means that the number of file descriptors in the process cannot be permitted to exceed `FD_SETSIZE`.

Threads:

- Demultiplexing using `select` is performed in one of the threads that calls `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- At most `min(64, IOV_MAX)` buffers may be transferred in a single operation.

Tru64

Demultiplexing mechanism:

- Uses `select` for demultiplexing. This means that the number of file descriptors in the process cannot be permitted to exceed `FD_SETSIZE`.

Threads:

- Demultiplexing using `select` is performed in one of the threads that calls `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- At most `min(64, IOV_MAX)` buffers may be transferred in a single operation.

Windows 95, 98 and Me

Demultiplexing mechanism:

- Uses `select` for demultiplexing.

Threads:

- Demultiplexing using `select` is performed in one of the threads that calls `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- For sockets, at most 16 buffers may be transferred in a single operation.

Windows NT, 2000, XP, 2003, Vista, 7 and 8

Demultiplexing mechanism:

- Uses overlapped I/O and I/O completion ports for all asynchronous socket operations except for asynchronous connect.
- Uses `select` for emulating asynchronous connect.

Threads:

- Demultiplexing using I/O completion ports is performed in all threads that call `io_service::run()`, `io_service::run_one()`, `io_service::poll()` or `io_service::poll_one()`.
- An additional thread per `io_service` is used to trigger timers. This thread is created on construction of the first `deadline_timer` or `deadline_timer_service` objects.
- An additional thread per `io_service` is used for the `select` demultiplexing. This thread is created on the first call to `async_connect()`.
- An additional thread per `io_service` is used to emulate asynchronous host resolution. This thread is created on the first call to either `ip::tcp::resolver::async_resolve()` or `ip::udp::resolver::async_resolve()`.

Scatter-Gather:

- For sockets, at most 64 buffers may be transferred in a single operation.
- For stream-oriented handles, only one buffer may be transferred in a single operation.

Windows Runtime

Asio provides limited support for the Windows Runtime. It requires that the language extensions be enabled. Due to the restricted facilities exposed by the Windows Runtime API, the support comes with the following caveats:

- The core facilities such as the `io_service`, `strand`, `buffers`, composed operations, timers, etc., should all work as normal.
- For sockets, only client-side TCP is supported.
- Explicit binding of a client-side TCP socket is not supported.
- The `cancel()` function is not supported for sockets. Asynchronous operations may only be cancelled by closing the socket.
- Operations that use `null_buffers` are not supported.
- Only `tcp::no_delay` and `socket_base::keep_alive` options are supported.
- Resolvers do not support service names, only numbers. I.e. you must use "80" rather than "http".
- Most resolver query flags have no effect.

Demultiplexing mechanism:

- Uses the `Windows::Networking::Sockets::StreamSocket` class to implement asynchronous TCP socket operations.

Threads:

- Event completions are delivered to the Windows thread pool and posted to the `io_service` for the handler to be executed.
- An additional thread per `io_service` is used to trigger timers. This thread is created on construction of the first timer objects.

Scatter-Gather:

- For sockets, at most one buffer may be transferred in a single operation.

2 Using Asio

Supported Platforms

The following platforms and compilers have been tested:

- Win32 and Win64 using Visual C++ 7.1 and Visual C++ 8.0.
- Win32 using MinGW.
- Win32 using Cygwin. (`__USE_W32_SOCKETS` must be defined.)
- 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.

The following platforms may also work:

- AIX 5.3 using XL C/C++ v9.
- HP-UX 11i v3 using patched aC++ A.06.14.
- QNX Neutrino 6.3 using g++ 3.3 or later.
- Solaris using Sun Studio 11 or later.
- Tru64 v5.1 using Compaq C++ v7.1.

Dependencies

The following libraries must be available in order to link programs that use Asio:

- Boost.Regex (optional) if you use any of the `read_until()` or `async_read_until()` overloads that take a `boost::regex` parameter.
- [OpenSSL](#) (optional) if you use Asio's SSL support.

Furthermore, some of the examples also require Boost.Date_Time or Boost.Serialization libraries.

Note

With MSVC or Borland C++ you may want to add `-DBOOST_DATE_TIME_NO_LIB` and `-DBOOST_REGEX_NO_LIB` to your project settings to disable autolinking of the Boost.Date_Time and Boost.Regex libraries respectively. Alternatively, you may choose to build these libraries and link to them.

Optional separate compilation

By default, Asio is a header-only library. However, some developers may prefer to build Asio using separately compiled source code. To do this, add `#include <asio/impl/src.hpp>` to one (and only one) source file in a program, then build the program with `ASIO_SEPARATE_COMPILATION` defined in the project/compiler settings. Alternatively, `ASIO_DYN_LINK` may be defined to build a separately-compiled Asio as part of a shared library.

If using Asio's SSL support, you will also need to add `#include <asio/ssl/impl/src.hpp>`.

Building the tests and examples on Linux or UNIX

If the boost directory (e.g. the directory called `boost_1_34_1`) is in the same directory as the asio source kit, then you may configure asio by simply going:

```
./configure
```

in the root directory of the asio source kit. Note that configure will always use the most recent boost version it knows about (i.e. 1.34.1) in preference to earlier versions, if there is more than one version present.

If the boost directory is in some other location, then you need to specify this directory when running configure:

```
./configure --with-boost=path_to_boost
```

When specifying the boost directory in this way you should ensure that you use an absolute path.

To build the examples, simply run `make` in the root directory of the asio source kit. To also build and run the unit tests, to confirm that asio is working correctly, run `make check`.

Building the tests and examples with MSVC

To build using the MSVC 7.1 or MSVC 8.0 command line compiler, perform the following steps in a Command Prompt window:

- If you are using a version of boost other than 1.34.1, or if the boost directory (i.e. the directory called `boost_1_34_1`) is not in the same directory as the asio source kit, then specify the location of boost by running a command similar to set `BOOSTDIR=path_to_boost`. Ensure that you specify an absolute path.
- Change to the asio `src` directory.
- Execute the command `nmake -f Makefile.msc`.
- Execute the command `nmake -f Makefile.msc check` to run a suite of tests to confirm that asio is working correctly.

Building the tests and examples with MinGW

To build using the MinGW g++ compiler from the command line, perform the following steps in a Command Prompt window:

- If you are using a version of boost other than 1.34.1, or if the boost directory (i.e. the directory called `boost_1_34_1`) is not in the same directory as the asio source kit, then specify the location of boost by running a command similar to set `BOOSTDIR=path_to_boost`. Ensure that you specify an absolute path using *forward slashes* (i.e. `c:/projects/boost_1_34_1` rather than `c:\projects\boost_1_34_1`).
- Change to the asio `src` directory.
- Execute the command `make -f Makefile.mgw`.
- Execute the command `make -f Makefile.mgw check` to run a suite of tests to confirm that asio is working correctly.

Note

The above instructions do not work when building inside MSYS. If you want to build using MSYS, you should use `export` rather than `set` to specify the location of boost.

Macros

The macros listed in the table below may be used to control the behaviour of Asio.

Macro	Description
ASIO_ENABLE_BUFFER_DEBUGGING	<p>Enables Asio's buffer debugging support, which can help identify when invalid buffers are used in read or write operations (e.g. if a std::string object being written is destroyed before the write operation completes).</p> <p>When using Microsoft Visual C++, this macro is defined automatically if the compiler's iterator debugging support is enabled, unless ASIO_DISABLE_BUFFER_DEBUGGING has been defined.</p> <p>When using g++, this macro is defined automatically if standard library debugging is enabled (_GLIBCXX_DEBUG is defined), unless ASIO_DISABLE_BUFFER_DEBUGGING has been defined.</p>
ASIO_DISABLE_BUFFER_DEBUGGING	Explicitly disables Asio's buffer debugging support.
ASIO_DISABLE_DEV_POLL	Explicitly disables /dev/poll support on Solaris, forcing the use of a select-based implementation.
ASIO_DISABLE_EPOLL	Explicitly disables epoll support on Linux, forcing the use of a select-based implementation.
ASIO_DISABLE_EVENTFD	Explicitly disables eventfd support on Linux, forcing the use of a pipe to interrupt blocked epoll/select system calls.
ASIO_DISABLE_KQUEUE	Explicitly disables kqueue support on Mac OS X and BSD variants, forcing the use of a select-based implementation.
ASIO_DISABLE_IOCP	Explicitly disables I/O completion ports support on Windows, forcing the use of a select-based implementation.
ASIO_DISABLE_THREADS	Explicitly disables Asio's threading support, independent of whether or not Boost supports threads.
ASIO_NO_WIN32_LEAN_AND_MEAN	<p>By default, Asio will automatically define WIN32_LEAN_AND_MEAN when compiling for Windows, to minimise the number of Windows SDK header files and features that are included. The presence of ASIO_NO_WIN32_LEAN_AND_MEAN prevents WIN32_LEAN_AND_MEAN from being defined.</p>
ASIO_NO_NOMINMAX	<p>By default, Asio will automatically define NOMINMAX when compiling for Windows, to suppress the definition of the min() and max() macros. The presence of ASIO_NO_NOMINMAX prevents NOMINMAX from being defined.</p>
ASIO_NO_DEFAULT_LINKED_LIBS	<p>When compiling for Windows using Microsoft Visual C++ or Borland C++, Asio will automatically link in the necessary Windows SDK libraries for sockets support (i.e. ws2_32.lib and mssock.lib, or ws2.lib when building for Windows CE). The ASIO_NO_DEFAULT_LINKED_LIBS macro prevents these libraries from being linked.</p>

Macro	Description
ASIO_SOCKET_STREAMBUF_MAX_ARITY	Determines the maximum number of arguments that may be passed to the <code>basic_socket_streambuf</code> class template's <code>connect</code> member function. Defaults to 5.
ASIO_SOCKET_IOSTREAM_MAX_ARITY	Determines the maximum number of arguments that may be passed to the <code>basic_socket_iostream</code> class template's constructor and <code>connect</code> member function. Defaults to 5.
ASIO_ENABLE_CANCELIO	<p>Enables use of the <code>CancelIo</code> function on older versions of Windows. If not enabled, calls to <code>cancel()</code> on a socket object will always fail with <code>asio::error::operation_not_supported</code> when run on Windows XP, Windows Server 2003, and earlier versions of Windows. When running on Windows Vista, Windows Server 2008, and later, the <code>CancelIoEx</code> function is always used.</p> <p>The <code>CancelIo</code> function has two issues that should be considered before enabling its use:</p> <ul style="list-style-type: none"> * It will only cancel asynchronous operations that were initiated in the current thread. * It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed. <p>For portable cancellation, consider using one of the following alternatives:</p> <ul style="list-style-type: none"> * Disable asio's I/O completion port backend by defining <code>ASIO_DISABLE_IOCP</code>. * Use the socket object's <code>close()</code> function to simultaneously cancel the outstanding operations and close the socket.
ASIO_NO_TYPEID	Disables uses of the <code>typeid</code> operator in asio. Defined automatically if <code>BOOST_NO_TYPEID</code> is defined.
ASIO_HASH_MAP_BUCKETS	<p>Determines the number of buckets in asio's internal <code>hash_map</code> objects. The value should be a comma separated list of prime numbers, in ascending order. The <code>hash_map</code> implementation will automatically increase the number of buckets as the number of elements in the map increases.</p> <p>Some examples:</p> <ul style="list-style-type: none"> * Defining <code>ASIO_HASH_MAP_BUCKETS</code> to 1021 means that the <code>hash_map</code> objects will always contain 1021 buckets, irrespective of the number of elements in the map. * Defining <code>ASIO_HASH_MAP_BUCKETS</code> to 53, 389, 1543 means that the <code>hash_map</code> objects will initially contain 53 buckets. The number of buckets will be increased to 389 and then 1543 as elements are added to the map.

Mailing List

A mailing list specifically for Asio may be found on SourceForge.net. Newsgroup access is provided via [Gmane](#).

Wiki

Users are encouraged to share examples, tips and FAQs on the Asio wiki, which is located at <http://think-async.com/Asio/>.

3 Tutorial

Basic Skills

The tutorial programs in this first section introduce the fundamental concepts required to use the asio toolkit. Before plunging into the complex world of network programming, these tutorial programs illustrate the basic skills using simple asynchronous timers.

- [Timer.1 - Using a timer synchronously](#)
- [Timer.2 - Using a timer asynchronously](#)
- [Timer.3 - Binding arguments to a handler](#)
- [Timer.4 - Using a member function as a handler](#)
- [Timer.5 - Synchronising handlers in multithreaded programs](#)

Introduction to Sockets

The tutorial programs in this section show how to use asio to develop simple client and server programs. These tutorial programs are based around the [daytime](#) protocol, which supports both TCP and UDP.

The first three tutorial programs implement the daytime protocol using TCP.

- [Daytime.1 - A synchronous TCP daytime client](#)
- [Daytime.2 - A synchronous TCP daytime server](#)
- [Daytime.3 - An asynchronous TCP daytime server](#)

The next three tutorial programs implement the daytime protocol using UDP.

- [Daytime.4 - A synchronous UDP daytime client](#)
- [Daytime.5 - A synchronous UDP daytime server](#)
- [Daytime.6 - An asynchronous UDP daytime server](#)

The last tutorial program in this section demonstrates how asio allows the TCP and UDP servers to be easily combined into a single program.

- [Daytime.7 - A combined TCP/UDP asynchronous server](#)

3.1 Timer.1 - Using a timer synchronously

This tutorial program introduces asio by showing how to perform a blocking wait on a timer.

We start by including the necessary header files.

All of the asio classes can be used by simply including the "asio.hpp" header file.

```
#include <iostream>
#include <asio.hpp>
```

Since this example uses timers, we need to include the appropriate Boost.Date_Time header file for manipulating times.

```
#include <boost/date_time posix_time posix_time.hpp>
```

All programs that use asio need to have at least one `io_service` object. This class provides access to I/O functionality. We declare an object of this type first thing in the main function.

```
int main()
{
    asio::io_service io;
```

Next we declare an object of type `asio::deadline_timer`. The core asio classes that provide I/O functionality (or as in this case timer functionality) always take a reference to an `io_service` as their first constructor argument. The second argument to the constructor sets the timer to expire 5 seconds from now.

```
    asio::deadline_timer t(io, boost::posix_time::seconds(5));
```

In this simple example we perform a blocking wait on the timer. That is, the call to `deadline_timer::wait()` will not return until the timer has expired, 5 seconds after it was created (i.e. not from when the wait starts).

A deadline timer is always in one of two states: "expired" or "not expired". If the `deadline_timer::wait()` function is called on an expired timer, it will return immediately.

```
    t.wait();
```

Finally we print the obligatory "Hello, world!" message to show when the timer has expired.

```
    std::cout << "Hello, world!" << std::endl;

    return 0;
}
```

See the [full source listing](#)

Return to the [tutorial index](#)

Next: [Timer.2 - Using a timer asynchronously](#)

3.1.1 Source listing for Timer.1

```
/*
// timer.cpp
// ~~~~~
//
// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)
//
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//

#include <iostream>
#include <asio.hpp>
#include <boost/date_time posix_time posix_time.hpp>

int main()
{
    asio::io_service io;

    asio::deadline_timer t(io, boost::posix_time::seconds(5));
    t.wait();

    std::cout << "Hello, world!" << std::endl;

    return 0;
}
```

Return to [Timer.1 - Using a timer synchronously](#)

3.2 Timer.2 - Using a timer asynchronously

This tutorial program demonstrates how to use asio's asynchronous callback functionality by modifying the program from tutorial Timer.1 to perform an asynchronous wait on the timer.

```
#include <iostream>
#include <asio.hpp>
#include <boost/date_time posix_time posix_time.hpp>
```

Using asio's asynchronous functionality means having a callback function that will be called when an asynchronous operation completes. In this program we define a function called `print` to be called when the asynchronous wait finishes.

```
void print(const asio::error_code& /*e*/)
{
    std::cout << "Hello, world!" << std::endl;
}

int main()
{
    asio::io_service io;

    asio::deadline_timer t(io, boost::posix_time::seconds(5));
```

Next, instead of doing a blocking wait as in tutorial Timer.1, we call the `deadline_timer::async_wait()` function to perform an asynchronous wait. When calling this function we pass the `print` callback handler that was defined above.

```
    t.async_wait(&print);
```

Finally, we must call the `io_service::run()` member function on the `io_service` object.

The asio library provides a guarantee that callback handlers will only be called from threads that are currently calling `io_service::run()`. Therefore unless the `io_service::run()` function is called the callback for the asynchronous wait completion will never be invoked.

The `io_service::run()` function will also continue to run while there is still "work" to do. In this example, the work is the asynchronous wait on the timer, so the call will not return until the timer has expired and the callback has completed.

It is important to remember to give the `io_service` some work to do before calling `io_service::run()`. For example, if we had omitted the above call to `deadline_timer::async_wait()`, the `io_service` would not have had any work to do, and consequently `io_service::run()` would have returned immediately.

```
    io.run();

    return 0;
}
```

See the [full source listing](#)

Return to the [tutorial index](#)

Previous: [Timer.1 - Using a timer synchronously](#)

Next: [Timer.3 - Binding arguments to a handler](#)

3.2.1 Source listing for Timer.2

```
//
// timer.cpp
// ~~~~~
//
// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)
//
// Distributed under the Boost Software License, Version 1.0. (See accompanying
```

```

// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//

#include <iostream>
#include <asio.hpp>
#include <boost/date_time posix_time posix_time.hpp>

void print(const asio::error_code& /*e*/)
{
    std::cout << "Hello, world!" << std::endl;
}

int main()
{
    asio::io_service io;

    asio::deadline_timer t(io, boost::posix_time::seconds(5));
    t.async_wait(&print);

    io.run();

    return 0;
}

```

Return to [Timer.2 - Using a timer asynchronously](#)

3.3 Timer.3 - Binding arguments to a handler

In this tutorial we will modify the program from tutorial Timer.2 so that the timer fires once a second. This will show how to pass additional parameters to your handler function.

```

#include <iostream>
#include <asio.hpp>
#include <boost/bind.hpp>
#include <boost/date_time posix_time posix_time.hpp>

```

To implement a repeating timer using asio you need to change the timer's expiry time in your callback function, and to then start a new asynchronous wait. Obviously this means that the callback function will need to be able to access the timer object. To this end we add two new parameters to the `print` function:

- A pointer to a timer object.
- A counter so that we can stop the program when the timer fires for the sixth time.

```

void print(const asio::error_code& /*e*/,
           asio::deadline_timer* t, int* count)
{

```

As mentioned above, this tutorial program uses a counter to stop running when the timer fires for the sixth time. However you will observe that there is no explicit call to ask the `io_service` to stop. Recall that in tutorial Timer.2 we learnt that the `io_service::run()` function completes when there is no more "work" to do. By not starting a new asynchronous wait on the timer when `count` reaches 5, the `io_service` will run out of work and stop running.

```

if (*count < 5)
{
    std::cout << *count << std::endl;
    ++(*count);
}

```

Next we move the expiry time for the timer along by one second from the previous expiry time. By calculating the new expiry time relative to the old, we can ensure that the timer does not drift away from the whole-second mark due to any delays in processing the handler.

```
t->expires_at(t->expires_at() + boost::posix_time::seconds(1));
```

Then we start a new asynchronous wait on the timer. As you can see, the `boost::bind` function is used to associate the extra parameters with your callback handler. The `deadline_timer::async_wait()` function expects a handler function (or function object) with the signature `void(const asio::error_code&)`. Binding the additional parameters converts your `print` function into a function object that matches the signature correctly.

See the [Boost.Bind documentation](#) for more information on how to use `boost::bind`.

In this example, the `asio::placeholders::error` argument to `boost::bind` is a named placeholder for the error object passed to the handler. When initiating the asynchronous operation, and if using `boost::bind`, you must specify only the arguments that match the handler's parameter list. In tutorial Timer.4 you will see that this placeholder may be elided if the parameter is not needed by the callback handler.

```
t->async_wait(boost::bind(print,
    asio::placeholders::error, t, count));
}

int main()
{
    asio::io_service io;
```

A new `count` variable is added so that we can stop the program when the timer fires for the sixth time.

```
int count = 0;
asio::deadline_timer t(io, boost::posix_time::seconds(1));
```

As in Step 4, when making the call to `deadline_timer::async_wait()` from `main` we bind the additional parameters needed for the `print` function.

```
t.async_wait(boost::bind(print,
    asio::placeholders::error, &t, &count));

io.run();
```

Finally, just to prove that the `count` variable was being used in the `print` handler function, we will print out its new value.

```
std::cout << "Final count is " << count << std::endl;

return 0;
}
```

See the [full source listing](#)

Return to the [tutorial index](#)

Previous: [Timer.2 - Using a timer asynchronously](#)

Next: [Timer.4 - Using a member function as a handler](#)

3.3.1 Source listing for Timer.3

```
//
// timer.cpp
// ~~~~~
//
// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)
```

```

// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//

#include <iostream>
#include <asio.hpp>
#include <boost/bind.hpp>
#include <boost/date_time posix_time posix_time.hpp>

void print(const asio::error_code& /*e*/,
           asio::deadline_timer* t, int* count)
{
    if (*count < 5)
    {
        std::cout << *count << std::endl;
        ++(*count);

        t->expires_at(t->expires_at() + boost::posix_time::seconds(1));
        t->async_wait(boost::bind(print,
                                  asio::placeholders::error, t, count));
    }
}

int main()
{
    asio::io_service io;

    int count = 0;
    asio::deadline_timer t(io, boost::posix_time::seconds(1));
    t.async_wait(boost::bind(print,
                            asio::placeholders::error, &t, &count));

    io.run();

    std::cout << "Final count is " << count << std::endl;

    return 0;
}

```

[Return to Timer.3 - Binding arguments to a handler](#)

3.4 Timer.4 - Using a member function as a handler

In this tutorial we will see how to use a class member function as a callback handler. The program should execute identically to the tutorial program from tutorial Timer.3.

```

#include <iostream>
#include <asio.hpp>
#include <boost/bind.hpp>
#include <boost/date_time posix_time posix_time.hpp>

```

Instead of defining a free function `print` as the callback handler, as we did in the earlier tutorial programs, we now define a class called `printer`.

```

class printer
{
public:

```

The constructor of this class will take a reference to the `io_service` object and use it when initialising the `timer_` member. The counter used to shut down the program is now also a member of the class.

```

printer(asio::io_service& io)
  : timer_(io, boost::posix_time::seconds(1)),
    count_(0)
{

```

The `boost::bind` function works just as well with class member functions as with free functions. Since all non-static class member functions have an implicit `this` parameter, we need to bind `this` to the function. As in tutorial Timer.3, `boost::bind` converts our callback handler (now a member function) into a function object that can be invoked as though it has the signature `void(const asio::error_code&)`.

You will note that the `asio::placeholders::error` placeholder is not specified here, as the `print` member function does not accept an error object as a parameter.

```

    timer_.async_wait(boost::bind(&printer::print, this));
}
```

In the class destructor we will print out the final value of the counter.

```

~printer()
{
    std::cout << "Final count is " << count_ << std::endl;
}
```

The `print` member function is very similar to the `print` function from tutorial Timer.3, except that it now operates on the class data members instead of having the timer and counter passed in as parameters.

```

void print()
{
    if (count_ < 5)
    {
        std::cout << count_ << std::endl;
        ++count_;

        timer_.expires_at(timer_.expires_at() + boost::posix_time::seconds(1));
        timer_.async_wait(boost::bind(&printer::print, this));
    }
}

private:
    asio::deadline_timer timer_;
    int count_;
};
```

The main function is much simpler than before, as it now declares a local `printer` object before running the `io_service` as normal.

```

int main()
{
    asio::io_service io;
    printer p(io);
    io.run();

    return 0;
}
```

See the [full source listing](#)

Return to the [tutorial index](#)

Previous: [Timer.3 - Binding arguments to a handler](#)

Next: [Timer.5 - Synchronising handlers in multithreaded programs](#)

3.4.1 Source listing for Timer.4

```
//  
// timer.cpp  
// ~~~~~  
//  
// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)  
//  
// Distributed under the Boost Software License, Version 1.0. (See accompanying  
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)  
//  
  
#include <iostream>  
#include <asio.hpp>  
#include <boost/bind.hpp>  
#include <boost/date_time posix_time posix_time.hpp>  
  
class printer  
{  
public:  
    printer(asio::io_service& io)  
        : timer_(io, boost::posix_time::seconds(1)),  
          count_(0)  
    {  
        timer_.async_wait(boost::bind(&printer::print, this));  
    }  
  
    ~printer()  
    {  
        std::cout << "Final count is " << count_ << std::endl;  
    }  
  
    void print()  
    {  
        if (count_ < 5)  
        {  
            std::cout << count_ << std::endl;  
            ++count_;  
  
            timer_.expires_at(timer_.expires_at() + boost::posix_time::seconds(1));  
            timer_.async_wait(boost::bind(&printer::print, this));  
        }  
    }  
  
private:  
    asio::deadline_timer timer_;  
    int count_;  
};  
  
int main()  
{  
    asio::io_service io;  
    printer p(io);  
    io.run();  
  
    return 0;  
}
```

Return to [Timer.4 - Using a member function as a handler](#)

3.5 Timer.5 - Synchronising handlers in multithreaded programs

This tutorial demonstrates the use of the `asio::strand` class to synchronise callback handlers in a multithreaded program.

The previous four tutorials avoided the issue of handler synchronisation by calling the `io_service::run()` function from one thread only. As you already know, the `asio` library provides a guarantee that callback handlers will only be called from threads that are currently calling `io_service::run()`. Consequently, calling `io_service::run()` from only one thread ensures that callback handlers cannot run concurrently.

The single threaded approach is usually the best place to start when developing applications using `asio`. The downside is the limitations it places on programs, particularly servers, including:

- Poor responsiveness when handlers can take a long time to complete.
- An inability to scale on multiprocessor systems.

If you find yourself running into these limitations, an alternative approach is to have a pool of threads calling `io_service::run()`. However, as this allows handlers to execute concurrently, we need a method of synchronisation when handlers might be accessing a shared, thread-unsafe resource.

```
#include <iostream>
#include <asio.hpp>
#include <boost/bind.hpp>
#include <boost/date_time posix_time posix_time.hpp>
```

We start by defining a class called `printer`, similar to the class in the previous tutorial. This class will extend the previous tutorial by running two timers in parallel.

```
class printer
{
public:
```

In addition to initialising a pair of `asio::deadline_timer` members, the constructor initialises the `strand_` member, an object of type `asio::strand`.

An `asio::strand` guarantees that, for those handlers that are dispatched through it, an executing handler will be allowed to complete before the next one is started. This is guaranteed irrespective of the number of threads that are calling `io_service::run()`. Of course, the handlers may still execute concurrently with other handlers that were not dispatched through an `asio::strand`, or were dispatched through a different `asio::strand` object.

```
printer(asio::io_service& io)
: strand_(io),
  timer1_(io, boost::posix_time::seconds(1)),
  timer2_(io, boost::posix_time::seconds(1)),
  count_(0)
{
```

When initiating the asynchronous operations, each callback handler is "wrapped" using the `asio::strand` object. The `strand::wrap()` function returns a new handler that automatically dispatches its contained handler through the `asio::strand` object. By wrapping the handlers using the same `asio::strand`, we are ensuring that they cannot execute concurrently.

```
    timer1_.async_wait(strand_.wrap(boost::bind(&printer::print1, this)));
    timer2_.async_wait(strand_.wrap(boost::bind(&printer::print2, this)));
}

~printer()
{
    std::cout << "Final count is " << count_ << std::endl;
}
```

In a multithreaded program, the handlers for asynchronous operations should be synchronised if they access shared resources. In this tutorial, the shared resources used by the handlers (`print1` and `print2`) are `std::cout` and the `count_` data member.

```

void print1()
{
    if (count_ < 10)
    {
        std::cout << "Timer 1: " << count_ << std::endl;
        ++count_;

        timer1_.expires_at(timer1_.expires_at() + boost::posix_time::seconds(1));
        timer1_.async_wait(strand_.wrap(boost::bind(&printer::print1, this)));
    }
}

void print2()
{
    if (count_ < 10)
    {
        std::cout << "Timer 2: " << count_ << std::endl;
        ++count_;

        timer2_.expires_at(timer2_.expires_at() + boost::posix_time::seconds(1));
        timer2_.async_wait(strand_.wrap(boost::bind(&printer::print2, this)));
    }
}

private:
    asio::io_service::strand strand_;
    asio::deadline_timer timer1_;
    asio::deadline_timer timer2_;
    int count_;
};

```

The `main` function now causes `io_service::run()` to be called from two threads: the main thread and one additional thread. This is accomplished using an `thread` object.

Just as it would with a call from a single thread, concurrent calls to `io_service::run()` will continue to execute while there is "work" left to do. The background thread will not exit until all asynchronous operations have completed.

```

int main()
{
    asio::io_service io;
    printer p(io);
    asio::thread t(boost::bind(&asio::io_service::run, &io));
    io.run();
    t.join();

    return 0;
}

```

See the [full source listing](#)

Return to the [tutorial index](#)

Previous: [Timer.4 - Using a member function as a handler](#)

3.5.1 Source listing for Timer.5

```

// 
// timer.cpp
// ~~~~~
// 
```

```

// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)
//
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//

#include <iostream>
#include <asio.hpp>
#include <boost/bind.hpp>
#include <boost/date_time posix_time posix_time.hpp>

class printer
{
public:
    printer(asio::io_service& io)
        : strand_(io),
          timer1_(io, boost::posix_time::seconds(1)),
          timer2_(io, boost::posix_time::seconds(1)),
          count_(0)
    {
        timer1_.async_wait(strand_.wrap(boost::bind(&printer::print1, this)));
        timer2_.async_wait(strand_.wrap(boost::bind(&printer::print2, this)));
    }

    ~printer()
    {
        std::cout << "Final count is " << count_ << std::endl;
    }

    void print1()
    {
        if (count_ < 10)
        {
            std::cout << "Timer 1: " << count_ << std::endl;
            ++count_;

            timer1_.expires_at(timer1_.expires_at() + boost::posix_time::seconds(1));
            timer1_.async_wait(strand_.wrap(boost::bind(&printer::print1, this)));
        }
    }

    void print2()
    {
        if (count_ < 10)
        {
            std::cout << "Timer 2: " << count_ << std::endl;
            ++count_;

            timer2_.expires_at(timer2_.expires_at() + boost::posix_time::seconds(1));
            timer2_.async_wait(strand_.wrap(boost::bind(&printer::print2, this)));
        }
    }

private:
    asio::io_service::strand strand_;
    asio::deadline_timer timer1_;
    asio::deadline_timer timer2_;
    int count_;
};

int main()
{

```

```

asio::io_service io;
printer p(io);
asio::thread t(boost::bind(&asio::io_service::run, &io));
io.run();
t.join();

return 0;
}

```

[Return to Timer.5 - Synchronising handlers in multithreaded programs](#)

3.6 Daytime.1 - A synchronous TCP daytime client

This tutorial program shows how to use asio to implement a client application with TCP.

We start by including the necessary header files.

```
#include <iostream>
#include <boost/array.hpp>
#include <asio.hpp>
```

The purpose of this application is to access a daytime service, so we need the user to specify the server.

```
using asio::ip::tcp;

int main(int argc, char* argv[])
{
    try
    {
        if (argc != 2)
        {
            std::cerr << "Usage: client <host>" << std::endl;
            return 1;
        }
    }
```

All programs that use asio need to have at least one `io_service` object.

```
asio::io_service io_service;
```

We need to turn the server name that was specified as a parameter to the application, into a TCP endpoint. To do this we use an `ip::tcp::resolver` object.

```
tcp::resolver resolver(io_service);
```

A resolver takes a query object and turns it into a list of endpoints. We construct a query using the name of the server, specified in `argv[1]`, and the name of the service, in this case "daytime".

```
tcp::resolver::query query(argv[1], "daytime");
```

The list of endpoints is returned using an iterator of type `ip::tcp::resolver::iterator`. (Note that a default constructed `ip::tcp::resolver::iterator` object can be used as an end iterator.)

```
tcp::resolver::iterator endpoint_iterator = resolver.resolve(query);
```

Now we create and connect the socket. The list of endpoints obtained above may contain both IPv4 and IPv6 endpoints, so we need to try each of them until we find one that works. This keeps the client program independent of a specific IP version. The `asio::connect()` function does this for us automatically.

```
tcp::socket socket(io_service);
asio::connect(socket, endpoint_iterator);
```

The connection is open. All we need to do now is read the response from the daytime service.

We use a `boost::array` to hold the received data. The `asio::buffer()` function automatically determines the size of the array to help prevent buffer overruns. Instead of a `boost::array`, we could have used a `char []` or `std::vector`.

```
for (;;)
{
    boost::array<char, 128> buf;
    asio::error_code error;

    size_t len = socket.read_some(asio::buffer(buf), error);
```

When the server closes the connection, the `ip::tcp::socket::read_some()` function will exit with the `asio::error::eof` error, which is how we know to exit the loop.

```
if (error == asio::error::eof)
    break; // Connection closed cleanly by peer.
else if (error)
    throw asio::system_error(error); // Some other error.

std::cout.write(buf.data(), len);
}
```

Finally, handle any exceptions that may have been thrown.

```
}
```

```
catch (std::exception& e)
{
    std::cerr << e.what() << std::endl;
}
```

See the [full source listing](#)

Return to the [tutorial index](#)

Next: [Daytime.2 - A synchronous TCP daytime server](#)

3.6.1 Source listing for Daytime.1

```
//
// client.cpp
// ~~~~~
//
// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)
//
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//

#include <iostream>
#include <boost/array.hpp>
#include <asio.hpp>

using asio::ip::tcp;

int main(int argc, char* argv[])
{
    try
    {
        if (argc != 2)
        {
```

```

    std::cerr << "Usage: client <host>" << std::endl;
    return 1;
}

asio::io_service io_service;

tcp::resolver resolver(io_service);
tcp::resolver::query query(argv[1], "daytime");
tcp::resolver::iterator endpoint_iterator = resolver.resolve(query);

tcp::socket socket(io_service);
asio::connect(socket, endpoint_iterator);

for (;;)
{
    boost::array<char, 128> buf;
    asio::error_code error;

    size_t len = socket.read_some(asio::buffer(buf), error);

    if (error == asio::error::eof)
        break; // Connection closed cleanly by peer.
    else if (error)
        throw asio::system_error(error); // Some other error.

    std::cout.write(buf.data(), len);
}
}

catch (std::exception& e)
{
    std::cerr << e.what() << std::endl;
}

return 0;
}

```

[Return to Daytime.1 - A synchronous TCP daytime client](#)

3.7 Daytime.2 - A synchronous TCP daytime server

This tutorial program shows how to use asio to implement a server application with TCP.

```

#include <ctime>
#include <iostream>
#include <string>
#include <asio.hpp>

using asio::ip::tcp;

```

We define the function `make_daytime_string()` to create the string to be sent back to the client. This function will be reused in all of our daytime server applications.

```

std::string make_daytime_string()
{
    using namespace std; // For time_t, time and ctime;
    time_t now = time(0);
    return ctime(&now);
}

int main()

```

```
{
    try
    {
        asio::io_service io_service;
```

A `ip::tcp::acceptor` object needs to be created to listen for new connections. It is initialised to listen on TCP port 13, for IP version 4.

```
        tcp::acceptor acceptor(io_service, tcp::endpoint(tcp::v4(), 13));
```

This is an iterative server, which means that it will handle one connection at a time. Create a socket that will represent the connection to the client, and then wait for a connection.

```
        for (;;)
        {
            tcp::socket socket(io_service);
            acceptor.accept(socket);
```

A client is accessing our service. Determine the current time and transfer this information to the client.

```
        std::string message = make_daytime_string();

        asio::error_code ignored_error;
        asio::write(socket, asio::buffer(message), ignored_error);
    }
}
```

Finally, handle any exceptions.

```
    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}
```

See the [full source listing](#)

Return to the [tutorial index](#)

Previous: [Daytime.1 - A synchronous TCP daytime client](#)

Next: [Daytime.3 - An asynchronous TCP daytime server](#)

3.7.1 Source listing for Daytime.2

```
/*
// server.cpp
// ~~~~~
//
// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)
//
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//

#include <ctime>
#include <iostream>
#include <string>
#include <asio.hpp>
```

```

using asio::ip::tcp;

std::string make_daytime_string()
{
    using namespace std; // For time_t, time and ctime;
    time_t now = time(0);
    return ctime(&now);
}

int main()
{
    try
    {
        asio::io_service io_service;

        tcp::acceptor acceptor(io_service, tcp::endpoint(tcp::v4(), 13));

        for (;;)
        {
            tcp::socket socket(io_service);
            acceptor.accept(socket);

            std::string message = make_daytime_string();

            asio::error_code ignored_error;
            asio::write(socket, asio::buffer(message), ignored_error);
        }
    }
    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}

```

[Return to Daytime.2 - A synchronous TCP daytime server](#)

3.8 Daytime.3 - An asynchronous TCP daytime server

The main() function

```

int main()
{
    try
    {

```

We need to create a server object to accept incoming client connections. The **io_service** object provides I/O services, such as sockets, that the server object will use.

```

        asio::io_service io_service;
        tcp_server server(io_service);

```

Run the **io_service** object so that it will perform asynchronous operations on your behalf.

```

        io_service.run();
    }
    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }
}

```

```

    }
    return 0;
}

```

The tcp_server class

```

class tcp_server
{
public:

```

The constructor initialises an acceptor to listen on TCP port 13.

```

tcp_server(asio::io_service& io_service)
    : acceptor_(io_service, tcp::endpoint(tcp::v4(), 13))
{
    start_accept();
}

private:

```

The function `start_accept()` creates a socket and initiates an asynchronous accept operation to wait for a new connection.

```

void start_accept()
{
    tcp_connection::pointer new_connection =
        tcp_connection::create(acceptor_.get_io_service());

    acceptor_.async_accept(new_connection->socket(),
        boost::bind(&tcp_server::handle_accept, this, new_connection,
           asio::placeholders::error));
}

```

The function `handle_accept()` is called when the asynchronous accept operation initiated by `start_accept()` finishes. It services the client request, and then calls `start_accept()` to initiate the next accept operation.

```

void handle_accept(tcp_connection::pointer new_connection,
    const asio::error_code& error)
{
    if (!error)
    {
        new_connection->start();
    }

    start_accept();
}

```

The tcp_connection class

We will use `shared_ptr` and `enable_shared_from_this` because we want to keep the `tcp_connection` object alive as long as there is an operation that refers to it.

```

class tcp_connection
    : public boost::enable_shared_from_this<tcp_connection>
{
public:
    typedef boost::shared_ptr<tcp_connection> pointer;

```

```

static pointer create(asio::io_service& io_service)
{
    return pointer(new tcp_connection(io_service));
}

tcp::socket& socket()
{
    return socket_;
}

```

In the function `start()`, we call `asio::async_write()` to serve the data to the client. Note that we are using `asio::async_write()`, rather than `ip::tcp::socket::async_write_some()`, to ensure that the entire block of data is sent.

```

void start()
{

```

The data to be sent is stored in the class member `message_` as we need to keep the data valid until the asynchronous operation is complete.

```
    message_ = make_daytime_string();
```

When initiating the asynchronous operation, and if using `boost::bind`, you must specify only the arguments that match the handler's parameter list. In this program, both of the argument placeholders (`asio::placeholders::error` and `asio::placeholders::bytes_transferred`) could potentially have been removed, since they are not being used in `handle_write()`.

```

asio::async_write(socket_,
    boost::bind(&tcp_connection::handle_write, shared_from_this(),
        asio::placeholders::error,
        asio::placeholders::bytes_transferred));

```

Any further actions for this client connection are now the responsibility of `handle_write()`.

```

}

private:
    tcp_connection(asio::io_service& io_service)
        : socket_(io_service)
    {
    }

    void handle_write(const asio::error_code& /*error*/,
        size_t /*bytes_transferred*/)
    {
    }

    tcp::socket socket_;
    std::string message_;
};
```

Removing unused handler parameters

You may have noticed that the `error`, and `bytes_transferred` parameters are not used in the body of the `handle_write()` function. If parameters are not needed, it is possible to remove them from the function so that it looks like:

```

void handle_write()
{
}
```

The `asio::async_write()` call used to initiate the call can then be changed to just:

```
asio::async_write(socket_, asio::buffer(message_),
    boost::bind(&tcp_connection::handle_write, shared_from_this()));
```

See the [full source listing](#)

Return to the [tutorial index](#)

Previous: [Daytime.2 - A synchronous TCP daytime server](#)

Next: [Daytime.4 - A synchronous UDP daytime client](#)

3.8.1 Source listing for Daytime.3

```
//  
// server.cpp  
// ~~~~~  
//  
// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)  
//  
// Distributed under the Boost Software License, Version 1.0. (See accompanying  
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)  
//  
  
#include <ctime>  
#include <iostream>  
#include <string>  
#include <boost/bind.hpp>  
#include <boost/shared_ptr.hpp>  
#include <boost/enable_shared_from_this.hpp>  
#include <asio.hpp>  
  
using asio::ip::tcp;  
  
std::string make_daytime_string()  
{  
    using namespace std; // For time_t, time and ctime;  
    time_t now = time(0);  
    return ctime(&now);  
}  
  
class tcp_connection  
    : public boost::enable_shared_from_this<tcp_connection>  
{  
public:  
    typedef boost::shared_ptr<tcp_connection> pointer;  
  
    static pointer create(asio::io_service& io_service)  
    {  
        return pointer(new tcp_connection(io_service));  
    }  
  
    tcp::socket& socket()  
    {  
        return socket_;  
    }  
  
    void start()  
    {  
        message_ = make_daytime_string();  
  
        asio::async_write(socket_, asio::buffer(message_),
```

```

        boost::bind(&tcp_connection::handle_write, shared_from_this(),
            asio::placeholders::error,
            asio::placeholders::bytes_transferred));
    }

private:
    tcp_connection(asio::io_service& io_service)
        : socket_(io_service)
    {
    }

void handle_write(const asio::error_code& /*error*/,
    size_t /*bytes_transferred*/)
{
}

tcp::socket socket_;
std::string message_;
};

class tcp_server
{
public:
    tcp_server(asio::io_service& io_service)
        : acceptor_(io_service, tcp::endpoint(tcp::v4(), 13))
    {
        start_accept();
    }

private:
    void start_accept()
    {
        tcp_connection::pointer new_connection =
            tcp_connection::create(acceptor_.get_io_service());

        acceptor_.async_accept(new_connection->socket(),
            boost::bind(&tcp_server::handle_accept, this, new_connection,
                asio::placeholders::error));
    }

    void handle_accept(tcp_connection::pointer new_connection,
        const asio::error_code& error)
    {
        if (!error)
        {
            new_connection->start();

            start_accept();
        }
    }

    tcp::acceptor acceptor_;
};

int main()
{
    try
    {
        asio::io_service io_service;
        tcp_server server(io_service);
        io_service.run();
    }
}

```

```

    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}

```

Return to [Daytime.3 - An asynchronous TCP daytime server](#)

3.9 Daytime.4 - A synchronous UDP daytime client

This tutorial program shows how to use asio to implement a client application with UDP.

```

#include <iostream>
#include <boost/array.hpp>
#include <asio.hpp>

using asio::ip::udp;

```

The start of the application is essentially the same as for the TCP daytime client.

```

int main(int argc, char* argv[])
{
    try
    {
        if (argc != 2)
        {
            std::cerr << "Usage: client <host>" << std::endl;
            return 1;
        }

        asio::io_service io_service;
    }

```

We use an `ip::udp::resolver` object to find the correct remote endpoint to use based on the host and service names. The query is restricted to return only IPv4 endpoints by the `ip::udp::v4()` argument.

```

        udp::resolver resolver(io_service);
        udp::resolver::query query(udp::v4(), argv[1], "daytime");

```

The `ip::udp::resolver::resolve()` function is guaranteed to return at least one endpoint in the list if it does not fail. This means it is safe to dereference the return value directly.

```

        udp::endpoint receiver_endpoint = *resolver.resolve(query);

```

Since UDP is datagram-oriented, we will not be using a stream socket. Create an `ip::udp::socket` and initiate contact with the remote endpoint.

```

        udp::socket socket(io_service);
        socket.open(udp::v4());

        boost::array<char, 1> send_buf = {{ 0 }};
        socket.send_to(asio::buffer(send_buf), receiver_endpoint);

```

Now we need to be ready to accept whatever the server sends back to us. The endpoint on our side that receives the server's response will be initialised by `ip::udp::socket::receive_from()`.

```

        boost::array<char, 128> recv_buf;
        udp::endpoint sender_endpoint;
        size_t len = socket.receive_from(

```

```

       asio::buffer(recv_buf), sender_endpoint);

    std::cout.write(recv_buf.data(), len);
}

```

Finally, handle any exceptions that may have been thrown.

```

catch (std::exception& e)
{
    std::cerr << e.what() << std::endl;
}

return 0;
}

```

See the [full source listing](#)

Return to the [tutorial index](#)

Previous: [Daytime.3 - An asynchronous TCP daytime server](#)

Next: [Daytime.5 - A synchronous UDP daytime server](#)

3.9.1 Source listing for Daytime.4

```

//  

// client.cpp  

// ~~~~~  

//  

// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)  

//  

// Distributed under the Boost Software License, Version 1.0. (See accompanying  

// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)  

//  

#include <iostream>  

#include <boost/array.hpp>  

#include <asio.hpp>  

using asio::ip::udp;  

int main(int argc, char* argv[])
{
    try
    {
        if (argc != 2)
        {
            std::cerr << "Usage: client <host>" << std::endl;
            return 1;
        }

        asio::io_service io_service;

        udp::resolver resolver(io_service);
        udp::resolver::query query(udp::v4(), argv[1], "daytime");
        udp::endpoint receiver_endpoint = *resolver.resolve(query);

        udp::socket socket(io_service);
        socket.open(udp::v4());

        boost::array<char, 1> send_buf = {{ 0 }};

```

```

socket.send_to(asio::buffer(send_buf), receiver_endpoint);

boost::array<char, 128> recv_buf;
udp::endpoint sender_endpoint;
size_t len = socket.receive_from(
    asio::buffer(recv_buf), sender_endpoint);

std::cout.write(recv_buf.data(), len);
}

catch (std::exception& e)
{
    std::cerr << e.what() << std::endl;
}

return 0;
}

```

Return to [Daytime.4 - A synchronous UDP daytime client](#)

3.10 Daytime.5 - A synchronous UDP daytime server

This tutorial program shows how to use asio to implement a server application with UDP.

```

int main()
{
    try
    {
        asio::io_service io_service;

```

Create an `ip::udp::socket` object to receive requests on UDP port 13.

```
    udp::socket socket(io_service, udp::endpoint(udp::v4(), 13));
```

Wait for a client to initiate contact with us. The `remote_endpoint` object will be populated by `ip::udp::socket::receive_from()`.

```

    for (;;)
    {
        boost::array<char, 1> recv_buf;
        udp::endpoint remote_endpoint;
        asio::error_code error;
        socket.receive_from(asio::buffer(recv_buf),
            remote_endpoint, 0, error);

        if (error && error != asio::error::message_size)
            throw asio::system_error(error);
    }
}

```

Determine what we are going to send back to the client.

```
    std::string message = make_daytime_string();
```

Send the response to the `remote_endpoint`.

```

        asio::error_code ignored_error;
        socket.send_to(asio::buffer(message),
            remote_endpoint, 0, ignored_error);
    }
}

```

Finally, handle any exceptions.

```

    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}

```

See the [full source listing](#)

Return to the [tutorial index](#)

Previous: [Daytime.4 - A synchronous UDP daytime client](#)

Next: [Daytime.6 - An asynchronous UDP daytime server](#)

3.10.1 Source listing for Daytime.5

```

//  

// server.cpp  

// ~~~~~  

//  

// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)  

//  

// Distributed under the Boost Software License, Version 1.0. (See accompanying  

// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)  

//  

#include <ctime>  

#include <iostream>  

#include <string>  

#include <boost/array.hpp>  

#include <asio.hpp>  

using asio::ip::udp;  

std::string make_daytime_string()  

{  

    using namespace std; // For time_t, time and ctime;  

    time_t now = time(0);  

    return ctime(&now);
}  

int main()
{
    try
    {
        asio::io_service io_service;  

        udp::socket socket(io_service, udp::endpoint(udp::v4(), 13));  

        for (;;)
        {
            boost::array<char, 1> recv_buf;  

            udp::endpoint remote_endpoint;  

            asio::error_code error;  

            socket.receive_from(asio::buffer(recv_buf),  

                remote_endpoint, 0, error);  

            if (error && error != asio::error::message_size)
                throw asio::system_error(error);
        }
    }
}
```

```

        std::string message = make_daytime_string();

        asio::error_code ignored_error;
        socket.send_to(asio::buffer(message),
                      remote_endpoint, 0, ignored_error);
    }
}

catch (std::exception& e)
{
    std::cerr << e.what() << std::endl;
}

return 0;
}

```

[Return to Daytime.5 - A synchronous UDP daytime server](#)

3.11 Daytime.6 - An asynchronous UDP daytime server

The main() function

```

int main()
{
    try
    {

```

Create a server object to accept incoming client requests, and run the `io_service` object.

```

        asio::io_service io_service;
        udp_server server(io_service);
        io_service.run();
    }

    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}

```

The `udp_server` class

```

class udp_server
{
public:

```

The constructor initialises a socket to listen on UDP port 13.

```

    udp_server(asio::io_service& io_service)
        : socket_(io_service, udp::endpoint(udp::v4(), 13))
    {
        start_receive();
    }

private:
    void start_receive()
    {

```

The function `ip::udp::socket::async_receive_from()` will cause the application to listen in the background for a new request. When such a request is received, the `io_service` object will invoke the `handle_receive()` function with two arguments: a value of type `error_code` indicating whether the operation succeeded or failed, and a `size_t` value `bytes_transferred` specifying the number of bytes received.

```
socket_.async_receive_from(
    asio::buffer(recv_buffer_), remote_endpoint_,
    boost::bind(&udp_server::handle_receive, this,
        asio::placeholders::error,
        asio::placeholders::bytes_transferred));
}
```

The function `handle_receive()` will service the client request.

```
void handle_receive(const asio::error_code& error,
    std::size_t /*bytes_transferred*/)
{
```

The `error` parameter contains the result of the asynchronous operation. Since we only provide the 1-byte `recv_buffer_` to contain the client's request, the `io_service` object would return an error if the client sent anything larger. We can ignore such an error if it comes up.

```
if (!error || error == asio::error::message_size)
{
```

Determine what we are going to send.

```
boost::shared_ptr<std::string> message(
    new std::string(make_daytime_string()));
```

We now call `ip::udp::socket::async_send_to()` to serve the data to the client.

```
socket_.async_send_to(asio::buffer(*message), remote_endpoint_,
    boost::bind(&udp_server::handle_send, this, message,
        asio::placeholders::error,
        asio::placeholders::bytes_transferred));
```

When initiating the asynchronous operation, and if using `boost::bind`, you must specify only the arguments that match the handler's parameter list. In this program, both of the argument placeholders (`asio::placeholders::error` and `asio::placeholders::bytes_transferred`) could potentially have been removed.

Start listening for the next client request.

```
start_receive();
```

Any further actions for this client request are now the responsibility of `handle_send()`.

```
}
```

The function `handle_send()` is invoked after the service request has been completed.

```
void handle_send(boost::shared_ptr<std::string> /*message*/,
    const asio::error_code& /*error*/,
    std::size_t /*bytes_transferred*/)
{
}

udp::socket socket_;
udp::endpoint remote_endpoint_;
boost::array<char, 1> recv_buffer_;
};
```

See the [full source listing](#)

Return to the [tutorial index](#)

Previous: [Daytime.5 - A synchronous UDP daytime server](#)

Next: [Daytime.7 - A combined TCP/UDP asynchronous server](#)

3.11.1 Source listing for Daytime.6

```
//  
// server.cpp  
// ~~~~~  
//  
// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)  
//  
// Distributed under the Boost Software License, Version 1.0. (See accompanying  
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)  
//  
  
#include <ctime>  
#include <iostream>  
#include <string>  
#include <boost/array.hpp>  
#include <boost/bind.hpp>  
#include <boost/shared_ptr.hpp>  
#include <asio.hpp>  
  
using asio::ip::udp;  
  
std::string make_daytime_string()  
{  
    using namespace std; // For time_t, time and ctime;  
    time_t now = time(0);  
    return ctime(&now);  
}  
  
class udp_server  
{  
public:  
    udp_server(asio::io_service& io_service)  
        : socket_(io_service, udp::endpoint(udp::v4(), 13))  
    {  
        start_receive();  
    }  
  
private:  
    void start_receive()  
    {  
        socket_.async_receive_from(  
            asio::buffer(recv_buffer_), remote_endpoint_,  
            boost::bind(&udp_server::handle_receive, this,  
                asio::placeholders::error,  
                asio::placeholders::bytes_transferred));  
    }  
  
    void handle_receive(const asio::error_code& error,  
        std::size_t /*bytes_transferred*/)  
    {  
        if (!error || error == asio::error::message_size)  
        {  
            boost::shared_ptr<std::string> message(  
                new std::string(recv_buffer_.data(),  
                recv_buffer_.size()));  
            std::cout << *message << std::endl;  
        }  
    }  
};
```

```

        new std::string(make_daytime_string()));

    socket_.async_send_to(asio::buffer(*message), remote_endpoint_,
        boost::bind(&udp_server::handle_send, this, message,
           asio::placeholders::error,
            asio::placeholders::bytes_transferred));

    start_receive();
}
}

void handle_send(boost::shared_ptr<std::string> /*message*/,
    const asio::error_code& /*error*/,
    std::size_t /*bytes_transferred*/)
{
}

udp::socket socket_;
udp::endpoint remote_endpoint_;
boost::array<char, 1> recv_buffer_;
};

int main()
{
    try
    {
        asio::io_service io_service;
        udp_server server(io_service);
        io_service.run();
    }
    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}

```

Return to [Daytime.6 - An asynchronous UDP daytime server](#)

3.12 Daytime.7 - A combined TCP/UDP asynchronous server

This tutorial program shows how to combine the two asynchronous servers that we have just written, into a single server application.

The main() function

```

int main()
{
    try
    {
        asio::io_service io_service;

```

We will begin by creating a server object to accept a TCP client connection.

```
tcp_server server1(io_service);
```

We also need a server object to accept a UDP client request.

```
udp_server server2(io_service);
```

We have created two lots of work for the `io_service` object to do.

```
    io_service.run();
}
catch (std::exception& e)
{
    std::cerr << e.what() << std::endl;
}

return 0;
}
```

The `tcp_connection` and `tcp_server` classes

The following two classes are taken from [Daytime.3](#).

```
class tcp_connection
    : public boost::enable_shared_from_this<tcp_connection>
{
public:
    typedef boost::shared_ptr<tcp_connection> pointer;

    static pointer create(asio::io_service& io_service)
    {
        return pointer(new tcp_connection(io_service));
    }

    tcp::socket& socket()
    {
        return socket_;
    }

    void start()
    {
        message_ = make_daytime_string();

        asio::async_write(socket_,
                          asio::buffer(message_),
                          boost::bind(&tcp_connection::handle_write, shared_from_this()));
    }

private:
    tcp_connection(asio::io_service& io_service)
        : socket_(io_service)
    {}

    void handle_write()
    {}

    tcp::socket socket_;
    std::string message_;
};

class tcp_server
{
public:
    tcp_server(asio::io_service& io_service)
        : acceptor_(io_service, tcp::endpoint(tcp::v4(), 13))
    {
        start_accept();
    }
};
```

```

    }

private:
    void start_accept()
    {
        tcp_connection::pointer new_connection =
            tcp_connection::create(acceptor_.get_io_service());

        acceptor_.async_accept(new_connection->socket(),
            boost::bind(&tcp_server::handle_accept, this, new_connection,
               asio::placeholders::error));
    }

    void handle_accept(tcp_connection::pointer new_connection,
        const asio::error_code& error)
    {
        if (!error)
        {
            new_connection->start();

            start_accept();
        }
    }

    tcp::acceptor acceptor_;
};


```

The udp_server class

Similarly, this next class is taken from the [previous tutorial step](#) .

```

class udp_server
{
public:
    udp_server(asio::io_service& io_service)
        : socket_(io_service, udp::endpoint(udp::v4(), 13))
    {
        start_receive();
    }

private:
    void start_receive()
    {
        socket_.async_receive_from(
            asio::buffer(recv_buffer_), remote_endpoint_,
            boost::bind(&udp_server::handle_receive, this,
                asio::placeholders::error));
    }

    void handle_receive(const asio::error_code& error)
    {
        if (!error || error == asio::error::message_size)
        {
            boost::shared_ptr<std::string> message(
                new std::string(make_daytime_string()));

            socket_.async_send_to(asio::buffer(*message), remote_endpoint_,
                boost::bind(&udp_server::handle_send, this, message));

            start_receive();
        }
    }
};


```

```

}

void handle_send(boost::shared_ptr<std::string> /*message*/)
{
}

udp::socket socket_;
udp::endpoint remote_endpoint_;
boost::array<char, 1> recv_buffer_;
};
```

See the [full source listing](#)

Return to the [tutorial index](#)

Previous: [Daytime.6 - An asynchronous UDP daytime server](#)

3.12.1 Source listing for Daytime.7

```

//  

// server.cpp  

// ~~~~~  

//  

// Copyright (c) 2003-2015 Christopher M. Kohlhoff (chris at kohlhoff dot com)  

//  

// Distributed under the Boost Software License, Version 1.0. (See accompanying  

// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)  

//  

#include <ctime>  

#include <iostream>  

#include <string>  

#include <boost/array.hpp>  

#include <boost/bind.hpp>  

#include <boost/shared_ptr.hpp>  

#include <boost/enable_shared_from_this.hpp>  

#include <asio.hpp>  

using asio::ip::tcp;  

using asio::ip::udp;  

std::string make_daytime_string()  

{  

    using namespace std; // For time_t, time and ctime;  

    time_t now = time(0);  

    return ctime(&now);
}  

class tcp_connection  

    : public boost::enable_shared_from_this<tcp_connection>
{  

public:  

    typedef boost::shared_ptr<tcp_connection> pointer;  

    static pointer create(asio::io_service& io_service)
    {  

        return pointer(new tcp_connection(io_service));
    }
  

    tcp::socket& socket()
    {
```

```

        return socket_;
    }

void start()
{
    message_ = make_daytime_string();

    asio::async_write(socket_, asio::buffer(message_),
                      boost::bind(&tcp_connection::handle_write, shared_from_this()));
}

private:
    tcp_connection(asio::io_service& io_service)
        : socket_(io_service)
    {
    }

void handle_write()
{
}

tcp::socket socket_;
std::string message_;
};

class tcp_server
{
public:
    tcp_server(asio::io_service& io_service)
        : acceptor_(io_service, tcp::endpoint(tcp::v4(), 13))
    {
        start_accept();
    }

private:
    void start_accept()
    {
        tcp_connection::pointer new_connection =
            tcp_connection::create(acceptor_.get_io_service());

        acceptor_.async_accept(new_connection->socket(),
                              boost::bind(&tcp_server::handle_accept, this, new_connection,
                                         asio::placeholders::error));
    }

    void handle_accept(tcp_connection::pointer new_connection,
                      const asio::error_code& error)
    {
        if (!error)
        {
            new_connection->start();
        }

        start_accept();
    }

    tcp::acceptor acceptor_;
};

class udp_server
{
public:

```

```

udp_server(asio::io_service& io_service)
    : socket_(io_service, udp::endpoint(udp::v4(), 13))
{
    start_receive();
}

private:
void start_receive()
{
    socket_.async_receive_from(
        asio::buffer(recv_buffer_), remote_endpoint_,
        boost::bind(&udp_server::handle_receive, this,
           asio::placeholders::error));
}

void handle_receive(const asio::error_code& error)
{
    if (!error || error == asio::error::message_size)
    {
        boost::shared_ptr<std::string> message(
            new std::string(make_daytime_string()));

        socket_.async_send_to(asio::buffer(*message), remote_endpoint_,
            boost::bind(&udp_server::handle_send, this, message));

        start_receive();
    }
}

void handle_send(boost::shared_ptr<std::string> /*message*/)
{
}

udp::socket socket_;
udp::endpoint remote_endpoint_;
boost::array<char, 1> recv_buffer_;
};

int main()
{
    try
    {
        asio::io_service io_service;
        tcp_server server1(io_service);
        udp_server server2(io_service);
        io_service.run();
    }
    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}

```

Return to [Daytime.7 - A combined TCP/UDP asynchronous server](#)

3.13 boost::bind

See the [Boost: bind.hpp documentation](#) for more information on how to use `boost::bind`.

4 Examples

-
-

4.1 C++03 Examples

Allocation

This example shows how to customise the allocation of memory associated with asynchronous operations.

- [./src/examples/cpp03/allocation/server.cpp](#)

Buffers

This example demonstrates how to create reference counted buffers that can be used with socket read and write operations.

- [./src/examples/cpp03/buffers/reference_counted.cpp](#)

Chat

This example implements a chat server and client. The programs use a custom protocol with a fixed length message header and variable length message body.

- [./src/examples/cpp03/chat/chat_message.hpp](#)
- [./src/examples/cpp03/chat/chat_client.cpp](#)
- [./src/examples/cpp03/chat/chat_server.cpp](#)

The following POSIX-specific chat client demonstrates how to use the `posix::stream_descriptor` class to perform console input and output.

- [./src/examples/cpp03/chat posix_chat_client.cpp](#)

Echo

A collection of simple clients and servers, showing the use of both synchronous and asynchronous operations.

- [./src/examples/cpp03/echo/async_tcp_echo_server.cpp](#)
- [./src/examples/cpp03/echo/async_udp_echo_server.cpp](#)
- [./src/examples/cpp03/echo/blocking_tcp_echo_client.cpp](#)
- [./src/examples/cpp03/echo/blocking_tcp_echo_server.cpp](#)
- [./src/examples/cpp03/echo/blocking_udp_echo_client.cpp](#)
- [./src/examples/cpp03/echo/blocking_udp_echo_server.cpp](#)

Fork

These POSIX-specific examples show how to use Asio in conjunction with the `fork()` system call. The first example illustrates the steps required to start a daemon process:

- `./src/examples/cpp03/fork/daemon.cpp`

The second example demonstrates how it is possible to fork a process from within a completion handler.

- `./src/examples/cpp03/fork/process_per_connection.cpp`

HTTP Client

Example programs implementing simple HTTP 1.0 clients. These examples show how to use the `read_until` and `async_read_until` functions.

- `./src/examples/cpp03/http/client/sync_client.cpp`
- `./src/examples/cpp03/http/client/async_client.cpp`

HTTP Server

This example illustrates the use of asio in a simple single-threaded server implementation of HTTP 1.0. It demonstrates how to perform a clean shutdown by cancelling all outstanding asynchronous operations.

- `./src/examples/cpp03/http/server/connection.cpp`
- `./src/examples/cpp03/http/server/connection.hpp`
- `./src/examples/cpp03/http/server/connection_manager.cpp`
- `./src/examples/cpp03/http/server/connection_manager.hpp`
- `./src/examples/cpp03/http/server/header.hpp`
- `./src/examples/cpp03/http/server/main.cpp`
- `./src/examples/cpp03/http/server/mime_types.cpp`
- `./src/examples/cpp03/http/server/mime_types.hpp`
- `./src/examples/cpp03/http/server/reply.cpp`
- `./src/examples/cpp03/http/server/reply.hpp`
- `./src/examples/cpp03/http/server/request.hpp`
- `./src/examples/cpp03/http/server/request_handler.cpp`
- `./src/examples/cpp03/http/server/request_handler.hpp`
- `./src/examples/cpp03/http/server/request_parser.cpp`
- `./src/examples/cpp03/http/server/request_parser.hpp`
- `./src/examples/cpp03/http/server/server.cpp`
- `./src/examples/cpp03/http/server/server.hpp`

HTTP Server 2

An HTTP server using an io_service-per-CPU design.

- `./src/examples/cpp03/http/server2/connection.cpp`
- `./src/examples/cpp03/http/server2/connection.hpp`
- `./src/examples/cpp03/http/server2/header.hpp`
- `./src/examples/cpp03/http/server2/io_service_pool.cpp`
- `./src/examples/cpp03/http/server2/io_service_pool.hpp`
- `./src/examples/cpp03/http/server2/main.cpp`
- `./src/examples/cpp03/http/server2/mime_types.cpp`
- `./src/examples/cpp03/http/server2/mime_types.hpp`
- `./src/examples/cpp03/http/server2/reply.cpp`
- `./src/examples/cpp03/http/server2/reply.hpp`
- `./src/examples/cpp03/http/server2/request.hpp`
- `./src/examples/cpp03/http/server2/request_handler.cpp`
- `./src/examples/cpp03/http/server2/request_handler.hpp`
- `./src/examples/cpp03/http/server2/request_parser.cpp`
- `./src/examples/cpp03/http/server2/request_parser.hpp`
- `./src/examples/cpp03/http/server2/server.cpp`
- `./src/examples/cpp03/http/server2/server.hpp`

HTTP Server 3

An HTTP server using a single io_service and a thread pool calling `io_service::run()`.

- `./src/examples/cpp03/http/server3/connection.cpp`
- `./src/examples/cpp03/http/server3/connection.hpp`
- `./src/examples/cpp03/http/server3/header.hpp`
- `./src/examples/cpp03/http/server3/main.cpp`
- `./src/examples/cpp03/http/server3/mime_types.cpp`
- `./src/examples/cpp03/http/server3/mime_types.hpp`
- `./src/examples/cpp03/http/server3/reply.cpp`
- `./src/examples/cpp03/http/server3/reply.hpp`
- `./src/examples/cpp03/http/server3/request.hpp`
- `./src/examples/cpp03/http/server3/request_handler.cpp`
- `./src/examples/cpp03/http/server3/request_handler.hpp`
- `./src/examples/cpp03/http/server3/request_parser.cpp`
- `./src/examples/cpp03/http/server3/request_parser.hpp`
- `./src/examples/cpp03/http/server3/server.cpp`
- `./src/examples/cpp03/http/server3/server.hpp`

HTTP Server 4

A single-threaded HTTP server implemented using stackless coroutines.

- `./src/examples/cpp03/http/server4/file_handler.cpp`
- `./src/examples/cpp03/http/server4/file_handler.hpp`
- `./src/examples/cpp03/http/server4/header.hpp`
- `./src/examples/cpp03/http/server4/main.cpp`
- `./src/examples/cpp03/http/server4/mime_types.cpp`
- `./src/examples/cpp03/http/server4/mime_types.hpp`
- `./src/examples/cpp03/http/server4/reply.cpp`
- `./src/examples/cpp03/http/server4/reply.hpp`
- `./src/examples/cpp03/http/server4/request.hpp`
- `./src/examples/cpp03/http/server4/request_parser.cpp`
- `./src/examples/cpp03/http/server4/request_parser.hpp`
- `./src/examples/cpp03/http/server4/server.cpp`
- `./src/examples/cpp03/http/server4/server.hpp`

ICMP

This example shows how to use raw sockets with ICMP to ping a remote host.

- `./src/examples/cpp03/icmp/ping.cpp`
- `./src/examples/cpp03/icmp/ipv4_header.hpp`
- `./src/examples/cpp03/icmp/icmp_header.hpp`

Invocation

This example shows how to customise handler invocation. Completion handlers are added to a priority queue rather than executed immediately.

- `./src/examples/cpp03/invocation/prioritised_handlers.cpp`

Iostreams

Two examples showing how to use `ip::tcp::iostream`.

- `./src/examples/cpp03/iostreams/daytime_client.cpp`
- `./src/examples/cpp03/iostreams/daytime_server.cpp`
- `./src/examples/cpp03/iostreams/http_client.cpp`

Multicast

An example showing the use of multicast to transmit packets to a group of subscribers.

- `./src/examples/cpp03/multicast/receiver.cpp`
- `./src/examples/cpp03/multicast/sender.cpp`

Serialization

This example shows how Boost.Serialization can be used with asio to encode and decode structures for transmission over a socket.

- `./src/examples/cpp03/serialization/client.cpp`
- `./src/examples/cpp03/serialization/connection.hpp`
- `./src/examples/cpp03/serialization/server.cpp`
- `./src/examples/cpp03/serialization/stock.hpp`

Services

This example demonstrates how to integrate custom functionality (in this case, for logging) into asio's `io_service`, and how to use a custom service with `basic_stream_socket<>`.

- `./src/examples/cpp03/services/basic_logger.hpp`
- `./src/examples/cpp03/services/daytime_client.cpp`
- `./src/examples/cpp03/services/logger.hpp`
- `./src/examples/cpp03/services/logger_service.cpp`
- `./src/examples/cpp03/services/logger_service.hpp`
- `./src/examples/cpp03/services/stream_socket_service.hpp`

SOCKS 4

Example client program implementing the SOCKS 4 protocol for communication via a proxy.

- `./src/examples/cpp03/socks4/sync_client.cpp`
- `./src/examples/cpp03/socks4/socks4.hpp`

SSL

Example client and server programs showing the use of the `ssl::stream<>` template with asynchronous operations.

- `./src/examples/cpp03/ssl/client.cpp`
- `./src/examples/cpp03/ssl/server.cpp`

Timeouts

A collection of examples showing how to cancel long running asynchronous operations after a period of time.

- `./src/examples/cpp03/timeouts/async_tcp_client.cpp`
- `./src/examples/cpp03/timeouts/blocking_tcp_client.cpp`
- `./src/examples/cpp03/timeouts/blocking_udp_client.cpp`
- `./src/examples/cpp03/timeouts/server.cpp`

Timers

Examples showing how to customise deadline_timer using different time types.

- `./src/examples/cpp03/timers/tick_count_timer.cpp`
- `./src/examples/cpp03/timers/time_t_timer.cpp`

Porthopper

Example illustrating mixed synchronous and asynchronous operations, and how to use Boost.Lambda with Asio.

- `./src/examples/cpp03/porthopper/protocol.hpp`
- `./src/examples/cpp03/porthopper/client.cpp`
- `./src/examples/cpp03/porthopper/server.cpp`

Nonblocking

Example demonstrating reactor-style operations for integrating a third-party library that wants to perform the I/O operations itself.

- `./src/examples/cpp03/nonblocking/third_party_lib.cpp`

Spawn

Example of using the `asio::spawn()` function, a wrapper around the [Boost.Coroutine](#) library, to implement a chain of asynchronous operations using stackful coroutines.

- `./src/examples/cpp03/spawn/echo_server.cpp`

UNIX Domain Sockets

Examples showing how to use UNIX domain (local) sockets.

- `./src/examples/cpp03/local/connect_pair.cpp`
- `./src/examples/cpp03/local/stream_server.cpp`
- `./src/examples/cpp03/local/stream_client.cpp`

Windows

An example showing how to use the Windows-specific function `TransmitFile` with Asio.

- [..../src/examples/cpp03/windows/transmit_file.cpp](#)

4.2 C++11 Examples

Allocation

This example shows how to customise the allocation of memory associated with asynchronous operations.

- [..../src/examples/cpp11/allocation/server.cpp \(diff to C++03\)](#)

Buffers

This example demonstrates how to create reference counted buffers that can be used with socket read and write operations.

- [..../src/examples/cpp11/buffers/reference_counted.cpp \(diff to C++03\)](#)

Chat

This example implements a chat server and client. The programs use a custom protocol with a fixed length message header and variable length message body.

- [..../src/examples/cpp11/chat/chat_message.hpp \(diff to C++03\)](#)
- [..../src/examples/cpp11/chat/chat_client.cpp \(diff to C++03\)](#)
- [..../src/examples/cpp11/chat/chat_server.cpp \(diff to C++03\)](#)

Echo

A collection of simple clients and servers, showing the use of both synchronous and asynchronous operations.

- [..../src/examples/cpp11/echo/async_tcp_echo_server.cpp \(diff to C++03\)](#)
- [..../src/examples/cpp11/echo/async_udp_echo_server.cpp \(diff to C++03\)](#)
- [..../src/examples/cpp11/echo/blocking_tcp_echo_client.cpp \(diff to C++03\)](#)
- [..../src/examples/cpp11/echo/blocking_tcp_echo_server.cpp \(diff to C++03\)](#)
- [..../src/examples/cpp11/echo/blocking_udp_echo_client.cpp \(diff to C++03\)](#)
- [..../src/examples/cpp11/echo/blocking_udp_echo_server.cpp \(diff to C++03\)](#)

Futures

This example demonstrates how to use `std::future` in conjunction with Asio's asynchronous operations.

- [..../src/examples/cpp11/futures/daytime_client.cpp](#)

HTTP Server

This example illustrates the use of asio in a simple single-threaded server implementation of HTTP 1.0. It demonstrates how to perform a clean shutdown by cancelling all outstanding asynchronous operations.

- [./src/examples/cpp11/http/server/connection.cpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/connection.hpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/connection_manager.cpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/connection_manager.hpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/header.hpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/main.cpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/mime_types.cpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/mime_types.hpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/reply.cpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/reply.hpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/request.hpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/request_handler.cpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/request_handler.hpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/request_parser.cpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/request_parser.hpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/server.cpp](#) (diff to C++03)
- [./src/examples/cpp11/http/server/server.hpp](#) (diff to C++03)

Spawn

Example of using the `asio::spawn()` function, a wrapper around the [Boost.Coroutine](#) library, to implement a chain of asynchronous operations using stackful coroutines.

- [./src/examples/cpp11/spawn/echo_server.cpp](#) (diff to C++03)

5 Reference

Core			
<p>Classes</p> <pre>const_buffer const_buffers_1 coroutine error_code invalid_service_owner io_service io_service::id io_service::service io_service::strand io_service::work mutable_buffer mutable_buffers_1 null_buffers service_already_exists streambuf system_error thread use_future_t yield_context</pre> <p>Class Templates</p> <pre>basic_io_object basic_streambuf basic_yield_context buffered_read_stream buffered_stream buffered_write_stream buffers_iterator</pre>	<p>Free Functions</p> <pre>add_service asio_handler_allocate asio_handler_deallocate asio_handler_invoke asio_handler_is_continuation async_read async_read_at async_read_until async_write async_write_at buffer buffer_cast buffer_copy buffer_size buffers_begin buffers_end has_service read read_at read_until spawn transfer_all transfer_at_least transfer_exactly use_service write write_at</pre>	<p>Special Values</p> <pre>use_future</pre> <p>Boost.Bind Placeholders</p> <pre>placeholders::bytes_transferred placeholders::error placeholders::iterator placeholders::signal_number</pre> <p>Error Codes</p> <pre>error::basic_errors error::netdb_errors error::addrinfo_errors error::misc_errors</pre> <p>Type Traits</p> <pre>async_result handler_type is_match_condition is_read_buffered is_write_buffered</pre>	<p>Type Requirements</p> <pre>Asynchronous operations AsyncRandomAccessReadDevice AsyncRandomAccessWriteDevice AsyncReadStream AsyncWriteStream CompletionHandler ConstBufferSequence ConvertibleToConstBuffer ConvertibleToMutableBuffer Handler IoObjectService MutableBufferSequence ReadHandler Service SyncRandomAccessReadDevice SyncRandomAccessWriteDevice SyncReadStream SyncWriteStream WriteHandler</pre>

Networking

Classes <pre>generic::datagram_protocol generic::datagram_protocol::endpoint generic::datagram_protocol::socket generic::raw_protocol generic::raw_protocol::endpoint generic::raw_protocol::socket generic::seq_packet_protocol generic::seq_packet_protocol::endpoint generic::seq_packet_protocol::socket generic::stream_protocol generic::stream_protocol::endpoint generic::stream_protocol::iostream generic::stream_protocol::socket ip::address ip::address_v4 ip::address_v6 ip::icmp ip::icmp::endpoint ip::icmp::resolver ip::icmpt::socket ip::resolver_query_base ip::tcp ip::tcp::acceptor ip::tcp::endpoint ip::tcp::iostream ip::tcp::resolver ip::tcp::socket ip::udp ip::udp::endpoint ip::udp::resolver ip::udp::socket socket_base</pre>	Free Functions <pre>async_connect connect ip::host_name</pre> Class Templates <pre>basic_datagram_socket basic_deadline_timer basic_raw_socket basic_seq_packet_socket basic_socket basic_socket_acceptor basic_socket_iostream basic_socket_streambuf basic_stream_socket generic::basic_endpoint ip::basic_endpoint ip::basic_resolver ip::basic_resolver_entry ip::basic_resolver_iterator ip::basic_resolver_query</pre> Services <pre>datagram_socket_service ip::resolver_service raw_socket_service seq_packet_socket_service socket_acceptor_service stream_socket_service</pre>	Socket Options <pre>ip::multicast::enable_loopback ip::multicast::hops ip::multicast::join_group ip::multicast::leave_group ip::multicast::outbound_interface ip::tcp::no_delay ip::unicast::hops ip::v6_only socket_base::broadcast socket_base::debug socket_base::do_not_route socket_base::enable_connection_aborted socket_base::keep_alive socket_base::linger socket_base::receive_buffer_size socket_base::receive_low_watermark socket_base::reuse_address socket_base::send_buffer_size socket_base::send_low_watermark</pre>	I/O Control Commands <pre>socket_base::bytes_readable socket_base::non_blocking_io</pre> Type Requirements <pre>AcceptHandler ComposedConnectHandler ConnectHandler DatagramSocketService Endpoint GettableSocketOption InternetProtocol IoControlCommand Protocol RawSocketService ResolveHandler ResolverService SeqPacketSocketService SettableSocketOption SocketAcceptorService SocketService StreamSocketService</pre>
---	--	---	---

Timers	SSL	Serial Ports	Signal Handling
Classes <pre>deadline_timer</pre> Class Templates <pre>basic_deadline_timer time_traits</pre> Services <pre>deadline_timer_service</pre> Type Requirements <pre>TimerService TimeTraits WaitHandler</pre>	Classes <pre>ssl::context ssl::context_base ssl::rfc2818_verification ssl::stream_base ssl::verify_context</pre> Class Templates <pre>ssl::stream</pre> Type Requirements <pre>BufferedHandshakeHandler HandshakeHandler ShutdownHandler</pre>	Classes <pre>serial_port serial_port_base</pre> Class Templates <pre>basic_serial_port</pre> Services <pre>serial_port_service</pre> Serial Port Options <pre>serial_port_base::baud_rate serial_port_base::flow_control serial_port_base::parity serial_port_base::stop_bits serial_port_base::character_size</pre> Type Requirements <pre>GettableSerialPortOption SerialPortService SettableSerialPortOption</pre>	Classes <pre>signal_set</pre> Class Templates <pre>basic_signal_set</pre> Services <pre>signal_set_service</pre> Type Requirements <pre>SignalSetService SignalHandler</pre>

POSIX-specific	Windows-specific	
<p>Classes</p> <p>local::stream_protocol local::stream_protocol::acceptor local::stream_protocol::endpoint local::stream_protocol::iostream local::stream_protocol::socket local::datagram_protocol local::datagram_protocol::endpoint local::datagram_protocol::socket posix::descriptor_base posix::stream_descriptor</p> <p>Free Functions</p> <p>local::connect_pair</p>	<p>Class Templates</p> <p>local::basic_endpoint posix::basic_descriptor posix::basic_stream_descriptor</p> <p>Services</p> <p>posix::stream_descriptor_service</p> <p>Type Requirements</p> <p>DescriptorService StreamDescriptorService</p>	<p>Classes</p> <p>windows::overlapped_ptr windows::random_access_handle windows::stream_handle</p> <p>Class Templates</p> <p>windows::basic_handle windows::basic_random_access_handle windows::basic_stream_handle</p> <p>Services</p> <p>windows::random_access_handle_service windows::stream_handle_service</p> <p>Type Requirements</p> <p>HandleService RandomAccessHandleService StreamHandleService</p>

5.1 Requirements on asynchronous operations

In Asio, an asynchronous operation is initiated by a function that is named with the prefix `async_`. These functions will be referred to as *initiating functions*.

All initiating functions in Asio 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 in Asio may call the application programming interface (API) provided by the operating system. If such an operating system API call results in an error, the handler will be invoked with a `const error_code` lvalue that evaluates to true. Otherwise the handler will 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 will be invoked with a value of type `error_code` that corresponds to the failure condition described by *POSIX* for that function, if any. Otherwise the handler will be invoked with an implementation-defined `error_code` value that reflects the operating system error.

Asynchronous operations will not fail with an error condition that indicates interruption by a signal (*POSIX* `EINTR`). Asynchronous operations will 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 `get_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 `get_io_service()` member function of the first argument to the initiating function.

Arguments to initiating functions will 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 will 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 `get_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 `get_io_service()` member function that returns the associated `io_service` object.

The `io_service` object associated with an asynchronous operation will 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`.

This implies that the handler must not be called directly from within the initiating function, even if the asynchronous operation completes immediately.

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 `asio_handler_allocate()`, `asio_handler_deallocate()`, and `asio_handler_invoke()` calls to the corresponding functions for the user-supplied handler. A bound completion handler meets the requirements for a **completion handler**.

For example, a bound completion handler for a `ReadHandler` may be implemented as follows:

```
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* asio_handler_allocate(size_t size,
                           bound_read_handler<ReadHandler>* this_handler)
{
    using asio::asio_handler_allocate;
    return asio_handler_allocate(size, &this_handler->handler_);
}
```

Asio 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.

```

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

template<class F, class ReadHandler>
void asio_handler_invoke(const F& f,
                        bound_read_handler<ReadHandler>* this_handler)
{
    using asio::asio_handler_invoke;
    asio_handler_invoke(f, &this_handler->handler_);
}

```

If the thread that initiates an asynchronous operation terminates before the associated handler is invoked, the behaviour is implementation-defined. Specifically, on *Windows* versions prior to Vista, unfinished operations are cancelled when the initiating thread exits.

The handler argument to an initiating function defines a handler identity. That is, the original handler argument and any copies of the handler argument will be considered equivalent. If the implementation needs to allocate storage for an asynchronous operation, the implementation will perform `asio_handler_allocate(size, &h)`, where `size` is the required size in bytes, and `h` is the handler. The implementation will perform `asio_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 `asio_handler_invoke`. Multiple storage blocks may be allocated for a single asynchronous operation.

Return type of an initiating function

By default, initiating functions return `void`. This is always the case when the handler is a function pointer, C++11 lambda, or a function object produced by `boost::bind` or `std::bind`.

For other types, the return type may be customised via a two-step process:

1. A specialisation of the `handler_type` template, which is used to determine the true handler type based on the asynchronous operation's handler's signature.
2. A specialisation of the `async_result` template, which is used both to determine the return type and to extract the return value from the handler.

These two templates have been specialised to provide support for `stackful coroutines` and the C++11 `std::future` class.

As an example, consider what happens when enabling `std::future` support by using the `asio::use_future` special value, as in:

```

std::future<std::size_t> length =
    my_socket.async_read_some(my_buffer, asio::use_future);

```

When a handler signature has the form:

```

void handler(error_code ec, result_type result);

```

the initiating function returns a `std::future` templated on `result_type`. In the above `async_read_some` example, this is `std::size_t`. If the asynchronous operation fails, the `error_code` is converted into a `system_error` exception and passed back to the caller through the future.

Where a handler signature has the form:

```

void handler(error_code ec);

```

the initiating function instead returns `std::future<void>`.

5.2 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`.

Examples

A free function as an accept handler:

```
void accept_handler(
    const asio::error_code& ec)
{
    ...
}
```

An accept handler function object:

```
struct accept_handler
{
    ...
    void operator() (
        const asio::error_code& ec)
    {
        ...
    }
    ...
};
```

A non-static class member function adapted to an accept handler using `bind()`:

```
void my_class::accept_handler(
    const asio::error_code& ec)
{
    ...
}
...
acceptor.async_accept(...,
    boost::bind(&my_class::accept_handler,
        this, asio::placeholders::error));
```

5.3 Buffer-oriented asynchronous random-access read device requirements

In the table below, `a` denotes an asynchronous random access read device object, `o` denotes an offset of type `boost::uint64_t`, `mb` denotes an object satisfying [mutable buffer sequence](#) requirements, and `h` denotes an object satisfying [read handler](#) requirements.

Table 1: Buffer-oriented asynchronous random-access read device requirements

operation	type	semantics, pre/post-conditions
<code>a.get_io_service();</code>	<code>io_service&</code>	Returns the <code>io_service</code> object through which the <code>async_read_some_at</code> handler <code>h</code> will be invoked.

Table 1: (continued)

operation	type	semantics, pre/post-conditions
<code>a.async_read_some_at(o, mb, h);</code>	void	<p>Initiates an asynchronous operation to read one or more bytes of data from the device <code>a</code> at the offset <code>o</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The <code>async_read_some_at</code> operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous read operation is invoked, <p>whichever comes first.</p> <p>If the total size of all buffers in the sequence <code>mb</code> 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.</p>

5.4 Buffer-oriented asynchronous random-access write device requirements

In the table below, `a` denotes an asynchronous write stream object, `o` denotes an offset of type `boost::uint64_t`, `cb` denotes an object satisfying [constant buffer sequence](#) requirements, and `h` denotes an object satisfying [write handler](#) requirements.

Table 2: Buffer-oriented asynchronous random-access write device requirements

operation	type	semantics, pre/post-conditions
<code>a.get_io_service();</code>	<code>io_service&</code>	Returns the <code>io_service</code> object through which the <code>async_write_some_at</code> handler <code>h</code> will be invoked.
<code>a.async_write_some_at(o, cb, h);</code>	<code>void</code>	<p>Initiates an asynchronous operation to write one or more bytes of data to the device <code>a</code> at offset <code>o</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The constant buffer sequence <code>cb</code> specifies memory where the data to be written is located. The <code>async_write_some_at</code> operation shall always write a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>cb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>cb</code> is destroyed, or — the handler for the asynchronous write operation is invoked, <p>whichever comes first.</p> <p>If the total size of all buffers in the sequence <code>cb</code> 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.</p>

5.5 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 3: Buffer-oriented asynchronous read stream requirements

operation	type	semantics, pre/post-conditions
a.get_io_service();	io_service&	Returns the <code>io_service</code> object through which the <code>async_read_some</code> handler <code>h</code> will be invoked.
a.async_read_some(mb, h);	void	<p>Initiates an asynchronous operation to read one or more bytes of data from the stream <code>a</code>. The operation is performed via the <code>io_service</code> object <code>a</code>. <code>get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The <code>async_read_some</code> operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous read operation is invoked, <p>whichever comes first.</p> <p>If the total size of all buffers in the sequence <code>mb</code> 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.</p>

5.6 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 4: Buffer-oriented asynchronous write stream requirements

operation	type	semantics, pre/post-conditions
<code>a.get_io_service();</code>	<code>io_service&</code>	Returns the <code>io_service</code> object through which the <code>async_write_some</code> handler <code>h</code> will be invoked.
<code>a.async_write_some(cb, h);</code>	<code>void</code>	<p>Initiates an asynchronous operation to write one or more bytes of data to the stream <code>a</code>. The operation is performed via the <code>io_service</code> object <code>a</code>. <code>get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The constant buffer sequence <code>cb</code> specifies memory where the data to be written is located. The <code>async_write_some</code> operation shall always write a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>cb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>cb</code> is destroyed, or — the handler for the asynchronous write operation is invoked, <p>whichever comes first.</p> <p>If the total size of all buffers in the sequence <code>cb</code> 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.</p>

5.7 Buffered handshake handler requirements

A buffered handshake handler must meet the requirements for a **handler**. A value `h` of a buffered handshake 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`.

Examples

A free function as a buffered handshake handler:

```
void handshake_handler(
    const asio::error_code& ec,
    std::size_t bytes_transferred)
{
    ...
}
```

A buffered handshake handler function object:

```
struct handshake_handler
{
    ...
    void operator()(
        const asio::error_code& ec,
        std::size_t bytes_transferred)
    {
        ...
    }
    ...
};
```

A non-static class member function adapted to a buffered handshake handler using `bind()`:

```
void my_class::handshake_handler(
    const asio::error_code& ec,
    std::size_t bytes_transferred)
{
    ...
}
...
socket.async_handshake(...,
    boost::bind(&my_class::handshake_handler,
        this, asio::placeholders::error,
        asio::placeholders::bytes_transferred));
```

5.8 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()`.

Examples

A free function as a completion handler:

```
void completion_handler()
{
    ...
}
```

A completion handler function object:

```
struct completion_handler
{
    ...
    void operator()()
```

```
{  
    ...  
}  
...  
};
```

A non-static class member function adapted to a completion handler using `bind()`:

```
void my_class::completion_handler()  
{  
    ...  
}  
...  
my_io_service.post(boost::bind(&my_class::completion_handler, this));
```

5.9 Composed connect handler requirements

A composed connect handler must meet the requirements for a [handler](#). A value `h` of a composed connect handler class should work correctly in the expression `h(ec, i)`, where `ec` is an lvalue of type `const error_code` and `i` is an lvalue of the type `Iterator` used in the corresponding `connect()` or `async_connect()` function.

Examples

A free function as a composed connect handler:

```
void connect_handler(  
    const asio::error_code& ec,  
    asio::ip::tcp::resolver::iterator iterator)  
{  
    ...  
}
```

A composed connect handler function object:

```
struct connect_handler  
{  
    ...  
    template <typename Iterator>  
    void operator()(  
        const asio::error_code& ec,  
        Iterator iterator)  
    {  
        ...  
    }  
    ...  
};
```

A non-static class member function adapted to a composed connect handler using `bind()`:

```
void my_class::connect_handler(  
    const asio::error_code& ec,  
    asio::ip::tcp::resolver::iterator iterator)  
{  
    ...  
}  
...  
asio::async_connect(...,  
    boost::bind(&my_class::connect_handler,  
        this, asio::placeholders::error,  
        asio::placeholders::iterator));
```

5.10 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`.

Examples

A free function as a connect handler:

```
void connect_handler(
    const asio::error_code& ec)
{
    ...
}
```

A connect handler function object:

```
struct connect_handler
{
    ...
    void operator() (
        const asio::error_code& ec)
    {
        ...
    }
    ...
};
```

A non-static class member function adapted to a connect handler using `bind()`:

```
void my_class::connect_handler(
    const asio::error_code& ec)
{
    ...
}
...
socket.async_connect(...,
    boost::bind(&my_class::connect_handler,
        this, asio::placeholders::error));
```

5.11 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 5: ConstBufferSequence requirements

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

Table 5: (continued)

expression	return type	assertion/note pre/post-condition
X(a);		<p>post: <code>equal_const_buffer_seq(a, X(a))</code> where the binary predicate <code>equal_const_buffer_seq</code> is defined as</p> <pre> 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); } </pre> <p>and the binary predicate <code>equal_buffer</code> is defined as</p> <pre> 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); } </pre>

Table 5: (continued)

expression	return type	assertion/note pre/post-condition
X u(a);		<p>post:</p> <pre>distance(a.begin(), a.end) == distance(u.begin(), u.end) && equal(a.begin(), a.end(), u.begin(), equal_buffer)</pre> <p>where the binary predicate <code>equal_buffer</code> is defined as</p> <pre>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); }</pre>
(&a)->~X();	void	note: the destructor is applied to every element of <code>a</code> ; all the memory is deallocated.
a.begin();	const_iterator or convertible to const_iterator	
a.end();	const_iterator or convertible to const_iterator	

5.12 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 6: ConvertibleToConstBuffer requirements

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

5.13 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 7: 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)

5.14 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 8: DatagramSocketService requirements

expression	return type	assertion/note pre/post-condition
a.receive(b, mb, f, ec);	size_t	<p>pre: a.is_open(b).</p> <p>Reads one or more bytes of data from a connected socket b.</p> <p>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.</p> <p>If successful, returns the number of bytes read. Otherwise returns 0.</p>
a.async_receive(b, mb, f, rh);	void	<p>pre: a.is_open(b).</p> <p>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.get_io_service() and behaves according to asynchronous operation requirements.</p> <p>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.</p> <p>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:</p> <ul style="list-style-type: none"> — the last copy of mb is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first.</p> <p>If the operation completes successfully, the ReadHandler object rh is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

Table 8: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.receive_from(b, mb, e, f, ec);</code>	<code>size_t</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Reads one or more bytes of data from an unconnected socket <code>b</code>.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>If successful, returns the number of bytes read. Otherwise returns 0.</p>

Table 8: (continued)

expression	return type	assertion/note pre/post-condition
<pre>a.async_receive_from(b, mb, e, f, rh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to read one or more bytes of data from an unconnected socket <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first.</p> <p>The program must ensure the object <code>e</code> is valid until the handler for the asynchronous operation is invoked.</p> <p>If the operation completes successfully, the <code>ReadHandler</code> object <code>rh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

Table 8: (continued)

expression	return type	assertion/note pre/post-condition
a.send(b, cb, f, ec);	size_t	<p>pre: a.is_open(b).</p> <p>Writes one or more bytes of data to a connected socket b.</p> <p>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.</p> <p>If successful, returns the number of bytes written. Otherwise returns 0.</p>
a.async_send(b, cb, f, wh);	void	<p>pre: a.is_open(b).</p> <p>Initiates an asynchronous operation to write one or more bytes of data to a connected socket b. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>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.</p> <p>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:</p> <ul style="list-style-type: none"> — the last copy of cb is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first.</p> <p>If the operation completes successfully, the <code>WriteHandler</code> object wh is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

Table 8: (continued)

expression	return type	assertion/note pre/post-condition
<pre>const typename Protocol:: endpoint& u = e; a.send_to(b, cb, u, f, ec) ;</pre>	size_t	<p>pre: a.is_open(b).</p> <p>Writes one or more bytes of data to an unconnected socket b.</p> <p>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.</p> <p>If successful, returns the number of bytes written. Otherwise returns 0.</p>

Table 8: (continued)

expression	return type	assertion/note pre/post-condition
<pre>const typename Protocol:: endpoint& u = e; a.async_send(b, cb, u, f, wh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to write one or more bytes of data to an unconnected socket <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>The implementation shall maintain one or more copies of <code>cb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>cb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first.</p> <p>If the operation completes successfully, the <code>WriteHandler</code> object <code>wh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

5.15 Descriptor service requirements

A descriptor service must meet the requirements for an **I/O object service** with support for movability, as well as the additional requirements listed below.

In the table below, `X` denotes a descriptor service class, `a` and `ao` denote values of type `X`, `b` and `c` denote values of type `X::implementation_type`, `n` denotes a value of type `X::native_handle_type`, `ec` denotes a value of type `error_code`, `i` denotes a value meeting **IoControlCommand** requirements, and `u` and `v` denote identifiers.

Table 9: DescriptorService requirements

expression	return type	assertion/note pre/post-condition
X::native_handle_type		The implementation-defined native representation of a descriptor. 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.move_construct(b, c);		From IoObjectService requirements. The underlying native representation is moved from c to b.
a.move_assign(b, ao, c);		From IoObjectService requirements. Implicitly cancels asynchronous operations associated with b, as if by calling a.close(b, ec). Then the underlying native representation is moved from c to b.
a.assign(b, 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 <code>error::operation_aborted</code> . post: !a.is_open(b).
a.native_handle(b);	X::native_handle_type	

Table 9: (continued)

expression	return type	assertion/note pre/post-condition
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 <code>error::operation_aborted</code> .
a.io_control(b, i, ec);	error_code	pre: a.is_open(b).

5.16 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, s denotes a size in bytes, and u denotes an identifier.

Table 10: Endpoint requirements

expression	type	assertion/note pre/post-conditions
X:::protocol_type	type meeting <code>protocol</code> requirements	
X u;		
X();		
a.protocol();	protocol_type	
a.data();	a pointer	Returns a pointer suitable for passing as the <code>address</code> argument to <i>POSIX</i> functions such as <code>accept()</code> , <code>getpeername()</code> , <code>getsockname()</code> and <code>recvfrom()</code> . The implementation shall perform a <code>reinterpret_cast</code> on the pointer to convert it to <code>sockaddr*</code> .
const X& u = a; u.data();	a pointer	Returns a pointer suitable for passing as the <code>address</code> argument to <i>POSIX</i> functions such as <code>connect()</code> , or as the <code>dest_addr</code> argument to <i>POSIX</i> functions such as <code>sendto()</code> . The implementation shall perform a <code>reinterpret_cast</code> on the pointer to convert it to <code>const sockaddr*</code> .

Table 10: (continued)

expression	type	assertion/note pre/post-conditions
a.size();	size_t	Returns a value suitable for passing as the <i>address_len</i> argument to <i>POSIX</i> functions such as <code>connect()</code> , or as the <i>dest_len</i> argument to <i>POSIX</i> functions such as <code>sendto()</code> , after appropriate integer conversion has been performed.
a.resize(s);		post: a.size() == s Passed the value contained in the <i>address_len</i> argument to <i>POSIX</i> functions such as <code>accept()</code> , <code>getpeername()</code> , <code>getsockname()</code> and <code>recvfrom()</code> , 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 <i>address_len</i> argument to <i>POSIX</i> functions such as <code>accept()</code> , <code>getpeername()</code> , <code>getsockname()</code> and <code>recvfrom()</code> , after appropriate integer conversion has been performed.

5.17 Gettable serial port option requirements

In the table below, X denotes a serial port option class, a denotes a value of X, ec denotes a value of type `error_code`, and s denotes a value of implementation-defined type `storage` (where `storage` is the type `DCB` on Windows and `termios` on *POSIX* platforms), and u denotes an identifier.

Table 11: GettableSerialPortOption requirements

expression	type	assertion/note pre/post-conditions
const storage& u =s; a.load(u, ec);	<code>error_code</code>	<p>Retrieves the value of the serial port option from the storage.</p> <p>If successful, sets <code>ec</code> such that <code>!ec</code> is true. If an error occurred, sets <code>ec</code> such that <code>!!ec</code> is true. Returns <code>ec</code>.</p>

5.18 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 12: GettableSocketOption requirements

expression	type	assertion/note pre/post-conditions
a.level(p);	int	Returns a value suitable for passing as the <i>level</i> argument to <i>POSIX getsockopt ()</i> (or equivalent).
a.name(p);	int	Returns a value suitable for passing as the <i>option_name</i> argument to <i>POSIX getsockopt ()</i> (or equivalent).
a.data(p);	a pointer, convertible to <code>void*</code>	Returns a pointer suitable for passing as the <i>option_value</i> argument to <i>POSIX getsockopt ()</i> (or equivalent).
a.size(p);	<code>size_t</code>	Returns a value suitable for passing as the <i>option_len</i> argument to <i>POSIX getsockopt ()</i> (or equivalent), after appropriate integer conversion has been performed.
a.resize(p, s);		post: a.size(p) == s. Passed the value contained in the <i>option_len</i> argument to <i>POSIX getsockopt ()</i> (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.

5.19 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 13: Handler requirements

expression	return type	assertion/note pre/post-conditions
<pre>using asio:: → asio_handler_allocate; void* asio_handler_allocate(s, & → h);</pre>		<p>Returns a pointer to a block of memory of size <i>s</i>. The pointer must satisfy the same alignment requirements as a pointer returned by <code>::operator new()</code>. Throws <code>bad_alloc</code> on failure.</p> <p>The <code>asio_handler_allocate()</code> function is located using argument-dependent lookup. The function <code>asio::asio_handler_allocate()</code> serves as a default if no user-supplied function is available.</p>
<pre>using asio:: → asio_handler_deallocate; asio_handler_deallocate(p, ← s, &h);</pre>		<p>Frees a block of memory associated with a pointer <i>p</i>, of at least size <i>s</i>, that was previously allocated using <code>asio_handler_allocate()</code>.</p> <p>The <code>asio_handler_deallocate()</code> function is located using argument-dependent lookup. The function <code>asio::asio_handler_deallocate()</code> serves as a default if no user-supplied function is available.</p>
<pre>using asio:: → asio_handler_invoke; asio_handler_invoke(f, &h) ← ;</pre>		<p>Causes the function object <i>f</i> to be executed as if by calling <code>f()</code>.</p> <p>The <code>asio_handler_invoke()</code> function is located using argument-dependent lookup. The function <code>asio::asio_handler_invoke()</code> serves as a default if no user-supplied function is available.</p>

5.20 Handle service requirements

A handle service must meet the requirements for an **I/O object service** with support for movability, as well as the additional requirements listed below.

In the table below, *X* denotes a handle service class, *a* and *ao* denote values of type *X*, *b* and *c* denote values of type *X::implementation_type*, *n* denotes a value of type *X::native_handle_type*, *ec* denotes a value of type *error_code*, and *u* and *v* denote identifiers.

Table 14: HandleService requirements

expression	return type	assertion/note pre/post-condition
X::native_handle_type		The implementation-defined native representation of a handle. 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.move_construct(b, c);		From IoObjectService requirements. The underlying native representation is moved from c to b.
a.move_assign(b, ao, c);		From IoObjectService requirements. Implicitly cancels asynchronous operations associated with b, as if by calling a.close(b, ec). Then the underlying native representation is moved from c to b.
a.assign(b, 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 <code>error::operation_aborted</code> . post: !a.is_open(b).
a.native_handle(b);	X::native_handle_type	

Table 14: (continued)

expression	return type	assertion/note pre/post-condition
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 <code>error::operation_aborted</code> .

5.21 SSL handshake handler requirements

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

Examples

A free function as a handshake handler:

```
void handshake_handler(
    const asio::error_code& ec)
{
    ...
}
```

A handshake handler function object:

```
struct handshake_handler
{
    ...
    void operator()(
        const asio::error_code& ec)
    {
        ...
    }
    ...
};
```

A non-static class member function adapted to a handshake handler using `bind()`:

```
void my_class::handshake_handler(
    const asio::error_code& ec)
{
    ...
}
...
ssl_stream.async_handshake(...,
    boost::bind(&my_class::handshake_handler,
        this, asio::placeholders::error));
```

5.22 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 15: 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).

5.23 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 16: IoControlCommand requirements

expression	type	assertion/note pre/post-conditions
a.name();	int	Returns a value suitable for passing as the <i>request</i> argument to <i>POSIX ioctl()</i> (or equivalent).
a.data();	a pointer, convertible to void*	

5.24 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 and ao denote values of type X, b and c denote values of type X::implementation_type, and u denotes an identifier.

Table 17: IoObjectService requirements

expression	return type	assertion/note pre/post-condition
X::implementation_type		

Table 17: (continued)

expression	return type	assertion/note pre/post-condition
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() or move_construct().
a.move_construct(b, c);		note: only required for I/O objects that support movability.
a.move_assign(b, ao, c);		note: only required for I/O objects that support movability.

5.25 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 18: 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).

Table 18: (continued)

expression	return type	assertion/note pre/post-condition
X(a);		<p>post: equalMutableBufferSeq(a, X(a)) where the binary predicate equalMutableBufferSeq is defined as</p> <pre data-bbox="1052 460 1565 665"> bool ← equalMutableBufferSeq(const X& x1, const X& x2 ←) { </pre> <p>return</p> <pre data-bbox="1052 633 1468 897"> distance(x1.begin(), ← x1.end()) == distance(x2.begin ← (), x2.end()) && equal(x1.begin ← (), x1.end(), x2.begin ← (), equalBuffer); }</pre> <p>and the binary predicate equalBuffer is defined as</p> <pre data-bbox="1052 992 1452 1393"> bool equalBuffer(const X::value_type& v1, const X::value_type& v2) { mutableBuffer b1(v1); mutableBuffer b2(v2); return bufferCast<const void ← *>(b1) == bufferCast<const ← void*>(b2) && bufferSize(b1) ← == bufferSize(b2); }</pre>

Table 18: (continued)

expression	return type	assertion/note pre/post-condition
X u(a);		<p>post:</p> <pre>distance(a.begin(), a.end) == distance(u.begin(), u.end) && equal(a.begin(), a.end(), u.begin(), equal_buffer)</pre> <p>where the binary predicate <code>equal_buffer</code> is defined as</p> <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>
(&a)->~X();	void	note: the destructor is applied to every element of <code>a</code> ; all the memory is deallocated.
a.begin();	const_iterator or convertible to const_iterator	
a.end();	const_iterator or convertible to const_iterator	

5.26 Object handle service requirements

An object handle service must meet the requirements for a [handle service](#), as well as the additional requirements listed below.

In the table below, `X` denotes an object handle 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`, and `wh` denotes a value meeting [WaitHandler](#) requirements.

Table 19: ObjectHandleService requirements

expression	return type	assertion/note pre/post-condition
<code>a.wait(b, ec);</code>	<code>error_code</code>	pre: <code>a.is_open(b)</code> . Synchronously waits for the object represented by handle <code>b</code> to become signalled.
<code>a.async_wait(b, wh);</code>	<code>void</code>	pre: <code>a.is_open(b)</code> . Initiates an asynchronous operation to wait for the object represented by handle <code>b</code> to become signalled. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements .

5.27 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 20: Protocol requirements

expression	return type	assertion/note pre/post-conditions
<code>X::endpoint</code>	type meeting <code>endpoint</code> requirements	
<code>a.family()</code>	<code>int</code>	Returns a value suitable for passing as the <code>domain</code> argument to <code>POSIX socket()</code> (or equivalent).
<code>a.type()</code>	<code>int</code>	Returns a value suitable for passing as the <code>type</code> argument to <code>POSIX socket()</code> (or equivalent).
<code>a.protocol()</code>	<code>int</code>	Returns a value suitable for passing as the <code>protocol</code> argument to <code>POSIX socket()</code> (or equivalent).

5.28 Random access handle service requirements

A random access handle service must meet the requirements for a [handle service](#), as well as the additional requirements listed below.

In the table below, X denotes a random access handle 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, o denotes an offset of type boost::uint64_t, 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 21: RandomAccessHandleService requirements

expression	return type	assertion/note pre/post-condition
a.read_some_at(b, o, mb, ec);	size_t	<p>pre: a.is_open(b).</p> <p>Reads one or more bytes of data from a handle b at offset o.</p> <p>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.</p> <p>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.</p>

Table 21: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.async_read_some_at(b, o, mb, rh);</code>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to read one or more bytes of data from a handle <code>b</code> at offset <code>o</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first. If the total size of all buffers in the sequence <code>mb</code> 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.</p> <p>If the operation completes successfully, the <code>ReadHandler</code> object <code>rh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

Table 21: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.write_some_at(b, o, cb, ec);</code>	<code>size_t</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Writes one or more bytes of data to a handle <code>b</code> at offset <code>o</code>.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>If successful, returns the number of bytes written. Otherwise returns 0. If the total size of all buffers in the sequence <code>cb</code> is 0, the function shall return 0 immediately.</p>

Table 21: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.async_write_some_at(b, o, cb, wh);</code>	<code>void</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to write one or more bytes of data to a handle <code>b</code> at offset <code>o</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>The implementation shall maintain one or more copies of <code>cb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>cb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first. If the total size of all buffers in the sequence <code>cb</code> is 0, the asynchronous operation shall complete immediately and pass 0 as the argument to the handler that specifies the number of bytes read.</p> <p>If the operation completes successfully, the <code>WriteHandler</code> object <code>wh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

5.29 Raw socket service requirements

A raw 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 raw 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 22: RawSocketService requirements

expression	return type	assertion/note pre/post-condition
<code>a.receive(b, mb, f, ec);</code>	<code>size_t</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Reads one or more bytes of data from a connected socket <code>b</code>.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>If successful, returns the number of bytes read. Otherwise returns 0.</p>

Table 22: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.async_receive(b, mb, f, rh);</code>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to read one or more bytes of data from a connected socket <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first.</p> <p>If the operation completes successfully, the <code>ReadHandler</code> object <code>rh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>
<code>a.receive_from(b, mb, e, f, ec);</code>	size_t	<p>pre: <code>a.is_open(b)</code>.</p> <p>Reads one or more bytes of data from an unconnected socket <code>b</code>.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>If successful, returns the number of bytes read. Otherwise returns 0.</p>

Table 22: (continued)

expression	return type	assertion/note pre/post-condition
<pre>a.async_receive_from(b, mb, e, f, rh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to read one or more bytes of data from an unconnected socket <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first.</p> <p>The program must ensure the object <code>e</code> is valid until the handler for the asynchronous operation is invoked.</p> <p>If the operation completes successfully, the <code>ReadHandler</code> object <code>rh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

Table 22: (continued)

expression	return type	assertion/note pre/post-condition
a.send(b, cb, f, ec);	size_t	<p>pre: a.is_open(b).</p> <p>Writes one or more bytes of data to a connected socket b.</p> <p>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.</p> <p>If successful, returns the number of bytes written. Otherwise returns 0.</p>
a.async_send(b, cb, f, wh);	void	<p>pre: a.is_open(b).</p> <p>Initiates an asynchronous operation to write one or more bytes of data to a connected socket b. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>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.</p> <p>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:</p> <ul style="list-style-type: none"> — the last copy of cb is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first.</p> <p>If the operation completes successfully, the <code>WriteHandler</code> object wh is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

Table 22: (continued)

expression	return type	assertion/note pre/post-condition
<pre>const typename Protocol:: endpoint& u = e; a.send_to(b, cb, u, f, ec) ;</pre>	size_t	<p>pre: a.is_open(b).</p> <p>Writes one or more bytes of data to an unconnected socket b.</p> <p>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.</p> <p>If successful, returns the number of bytes written. Otherwise returns 0.</p>

Table 22: (continued)

expression	return type	assertion/note pre/post-condition
<pre>const typename Protocol:: endpoint& u = e; a.async_send(b, cb, u, f, wh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to write one or more bytes of data to an unconnected socket <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>The implementation shall maintain one or more copies of <code>cb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>cb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first.</p> <p>If the operation completes successfully, the <code>WriteHandler</code> object <code>wh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

5.30 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`.

Examples

A free function as a read handler:

```
void read_handler(
    const asio::error_code& ec,
    std::size_t bytes_transferred)
{
```

```
    ...
}
```

A read handler function object:

```
struct read_handler
{
    ...
    void operator()(const asio::error_code& ec,
                    std::size_t bytes_transferred)
    {
        ...
    }
    ...
};
```

A non-static class member function adapted to a read handler using `bind()`:

```
void my_class::read_handler(
    const asio::error_code& ec,
    std::size_t bytes_transferred)
{
    ...
}
...
socket.async_read(...,
    boost::bind(&my_class::read_handler,
               this, asio::placeholders::error,
               asio::placeholders::bytes_transferred));
```

5.31 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 `i` 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.

Examples

A free function as a resolve handler:

```
void resolve_handler(
    const asio::error_code& ec,
    asio::ip::tcp::resolver::iterator iterator)
{
    ...
}
```

A resolve handler function object:

```
struct resolve_handler
{
    ...
    void operator()(const asio::error_code& ec,
                    asio::ip::tcp::resolver::iterator iterator)
    {
        ...
    }
};
```

```

}
...
};
```

A non-static class member function adapted to a resolve handler using bind():

```

void my_class::resolve_handler(
    const asio::error_code& ec,
    asio::ip::tcp::resolver::iterator iterator)
{
    ...
}
...
resolver.async_resolve(...,
    boost::bind(&my_class::resolve_handler,
        this, asio::placeholders::error,
        asio::placeholders::iterator));
```

5.32 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 23: 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 <code>error::operation_aborted</code> .
a.resolve(b, q, ec);	ip::basic_resolver_iterator<InternetProtocol>	On success, returns an iterator i such that <code>i != ip::basic_resolver_iterator<InternetProtocol>()</code> . Otherwise returns <code>ip::basic_resolver_iterator<InternetProtocol>()</code> .

Table 23: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.async_resolve(b, q, h);</code>		<p>Initiates an asynchronous resolve operation that is performed via the <code>io_service</code> object <code>a</code>. <code>get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>If the operation completes successfully, the <code>ResolveHandler</code> object <code>h</code> shall be invoked with an iterator object <code>i</code> such that the condition <code>i !=ip::basic_resolver_iterator<InternetProtocol>()</code> holds. Otherwise it is invoked with <code>ip::basic_resolver_iterator<InternetProtocol>()</code>.</p>
<code>a.resolve(b, e, ec);</code>	<code>ip::basic_resolver_iterator<InternetProtocol></code>	<p>On success, returns an iterator <code>i</code> such that <code>i !=ip::basic_resolver_iterator<InternetProtocol>()</code>. Otherwise returns <code>ip::basic_resolver_iterator<InternetProtocol>()</code>.</p>
<code>a.async_resolve(b, e, h);</code>		<p>Initiates an asynchronous resolve operation that is performed via the <code>io_service</code> object <code>a</code>. <code>get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>If the operation completes successfully, the <code>ResolveHandler</code> object <code>h</code> shall be invoked with an iterator object <code>i</code> such that the condition <code>i !=ip::basic_resolver_iterator<InternetProtocol>()</code> holds. Otherwise it is invoked with <code>ip::basic_resolver_iterator<InternetProtocol>()</code>.</p>

5.33 Sequenced packet socket service requirements

A sequenced packet 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`, `g` denotes an lvalue 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 24: StreamSocketService requirements

expression	return type	assertion/note pre/post-condition
<code>a.receive(b, mb, f, g, ec);</code>	<code>size_t</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Reads one or more bytes of data from a connected socket b.</p> <p>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.</p> <p>If successful, sets g to the flags associated with the received data, and returns the number of bytes read. Otherwise, sets g to 0 and returns 0.</p>

Table 24: (continued)

expression	return type	assertion/note pre/post-condition
<pre>a.async_receive(b, mb, f, g, rh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to read one or more bytes of data from a connected socket <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first.</p> <p>If the operation completes successfully, sets <code>g</code> to the flags associated with the received data, then invokes the <code>ReadHandler</code> object <code>rh</code> with the number of bytes transferred. Otherwise, sets <code>g</code> to 0 and invokes <code>rh</code> with 0 bytes transferred.</p>

Table 24: (continued)

expression	return type	assertion/note pre/post-condition
a.send(b, cb, f, ec);	size_t	<p>pre: a.is_open(b).</p> <p>Writes one or more bytes of data to a connected socket b.</p> <p>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.</p> <p>If successful, returns the number of bytes written. Otherwise returns 0.</p>

Table 24: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.async_send(b, cb, f, wh);</code>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to write one or more bytes of data to a connected socket <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>The implementation shall maintain one or more copies of <code>cb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>cb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first.</p> <p>If the operation completes successfully, the <code>WriteHandler</code> object <code>wh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

5.34 Serial port service requirements

A serial port service must meet the requirements for an **I/O object service** with support for movability, as well as the additional requirements listed below.

In the table below, `X` denotes a serial port service class, `a` and `ao` denote values of type `X`, `d` denotes a serial port device name of type `std::string`, `b` and `c` denote values of type `X::implementation_type`, `n` denotes a value of type `X::native_handle_type`, `ec` denotes a value of type `error_code`, `s` denotes a value meeting `SettableSerialPortOption` requirements, `g` denotes a value meeting `GettableSerialPortOption` requirements, `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. `u` and `v` denote identifiers.

Table 25: SerialPortService requirements

expression	return type	assertion/note pre/post-condition
X::native_handle_type		The implementation-defined native representation of a serial port. 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.move_construct(b, c);		From IoObjectService requirements. The underlying native representation is moved from c to b.
a.move_assign(b, ao, c);		From IoObjectService requirements. Implicitly cancels asynchronous operations associated with b, as if by calling a.close(b, ec). Then the underlying native representation is moved from c to b.
const std::string& u = d; a.open(b, u, ec);	error_code	pre: !a.is_open(b). post: !!ec a.is_open(b).
a.assign(b, 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 <code>error::operation_aborted</code> . post: !a.is_open(b).

Table 25: (continued)

expression	return type	assertion/note pre/post-condition
a.native_handle(b);	X::native_handle_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 <code>error::operation_aborted</code> .
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.send_break(b, ec);	error_code	pre: a.is_open(b).
a.read_some(b, mb, ec);	size_t	pre: a.is_open(b). Reads one or more bytes of data from a serial port 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 <code>error::eof</code> .

Table 25: (continued)

expression	return type	assertion/note pre/post-condition
<pre>a.async_read_some(b, mb, rh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to read one or more bytes of data from a serial port <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first. If the total size of all buffers in the sequence <code>mb</code> 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.</p> <p>If the operation completes due to graceful connection closure by the peer, the operation shall fail with <code>error::eof</code>.</p> <p>If the operation completes successfully, the <code>ReadHandler</code> object <code>rh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

Table 25: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.write_some(b, cb, ec);</code>	<code>size_t</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Writes one or more bytes of data to a serial port <code>b</code>.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>If successful, returns the number of bytes written. Otherwise returns 0. If the total size of all buffers in the sequence <code>cb</code> is 0, the function shall return 0 immediately.</p>

Table 25: (continued)

expression	return type	assertion/note pre/post-condition
a.async_write_some(b, cb, wh);	void	<p>pre: a.is_open(b).</p> <p>Initiates an asynchronous operation to write one or more bytes of data to a serial port b. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>The implementation shall maintain one or more copies of <code>cb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>cb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first. If the total size of all buffers in the sequence <code>cb</code> is 0, the asynchronous operation shall complete immediately and pass 0 as the argument to the handler that specifies the number of bytes read.</p> <p>If the operation completes successfully, the <code>WriteHandler</code> object <code>wh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

5.35 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.

5.36 Settable serial port option requirements

In the table below, X denotes a serial port option class, a denotes a value of X, ec denotes a value of type `error_code`, and s denotes a value of implementation-defined type `storage` (where `storage` is the type `DCB` on Windows and `termios` on *POSIX* platforms), and u denotes an identifier.

Table 26: SettableSerialPortOption requirements

expression	type	assertion/note pre/post-conditions
<code>const X& u =a;</code> <code>u.store(s, ec);</code>	<code>error_code</code>	Saves the value of the serial port option to the storage. If successful, sets <code>ec</code> such that <code>!ec</code> is true. If an error occurred, sets <code>ec</code> such that <code>!ec</code> is true. Returns <code>ec</code> .

5.37 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 27: SettableSocketOption requirements

expression	type	assertion/note pre/post-conditions
<code>a.level(p);</code>	<code>int</code>	Returns a value suitable for passing as the <code>level</code> argument to <i>POSIX setsockopt()</i> (or equivalent).
<code>a.name(p);</code>	<code>int</code>	Returns a value suitable for passing as the <code>option_name</code> argument to <i>POSIX setsockopt()</i> (or equivalent).

Table 27: (continued)

expression	type	assertion/note pre/post-conditions
<code>const X& u =a;u.data(p);</code>	a pointer, convertible to <code>const void*</code>	Returns a pointer suitable for passing as the <code>option_value</code> argument to <code>POSIX setsockopt()</code> (or equivalent).
<code>a.size(p);</code>	<code>size_t</code>	Returns a value suitable for passing as the <code>option_len</code> argument to <code>POSIX setsockopt()</code> (or equivalent), after appropriate integer conversion has been performed.

5.38 SSL shutdown handler requirements

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

Examples

A free function as a shutdown handler:

```
void shutdown_handler(
    const asio::error_code& ec)
{
    ...
}
```

A shutdown handler function object:

```
struct shutdown_handler
{
    ...
    void operator()(

        const asio::error_code& ec)
    {
        ...
    }
    ...
};
```

A non-static class member function adapted to a shutdown handler using `bind()`:

```
void my_class::shutdown_handler(
    const asio::error_code& ec)
{
    ...
}
...
ssl_stream.async_shutdown(
    boost::bind(&my_class::shutdown_handler,
        this, asio::placeholders::error));
```

5.39 Signal handler requirements

A signal handler must meet the requirements for a [handler](#). A value `h` of a signal handler class should work correctly in the expression `h(ec, n)`, where `ec` is an lvalue of type `const error_code` and `n` is an lvalue of type `const int`.

Examples

A free function as a signal handler:

```
void signal_handler(
    const asio::error_code& ec,
    int signal_number)
{
    ...
}
```

A signal handler function object:

```
struct signal_handler
{
    ...
    void operator()(
        const asio::error_code& ec,
        int signal_number)
    {
        ...
    }
    ...
};
```

A non-static class member function adapted to a signal handler using `bind()`:

```
void my_class::signal_handler(
    const asio::error_code& ec,
    int signal_number)
{
    ...
}
...
my_signal_set.async_wait(
    boost::bind(&my_class::signal_handler,
        this, asio::placeholders::error,
        asio::placeholders::signal_number));
```

5.40 Signal set service requirements

A signal set 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 signal set 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`, `n` denotes a value of type `int`, and `sh` denotes a value meeting [SignalHandler](#) requirements.

Table 28: SignalSetService requirements

expression	return type	assertion/note pre/post-condition
a.construct(b);		From IoObjectService requirements.
a.destroy(b);		From IoObjectService requirements. Implicitly clears the registered signals as if by calling a.clear(b, ec), then implicitly cancels outstanding asynchronous operations as if by calling a.cancel(b, ec).
a.add(b, n, ec);	error_code	
a.remove(b, n, ec);	error_code	
a.clear(b, ec);	error_code	
a.cancel(b, ec);	error_code	
a.async_wait(b, sh);	void	<p>pre: a.is_open(b).</p> <p>Initiates an asynchronous operation to wait for the delivery of one of the signals registered for the signal set b. The operation is performed via the io_service object a.get_io_service() and behaves according to asynchronous operation requirements.</p> <p>If the operation completes successfully, the SignalHandler object sh is invoked with the number identifying the delivered signal. Otherwise it is invoked with 0.</p>

5.41 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 and ao denote values of type X, b and c denote values of type X::implementation_type, p denotes a value of type Protocol, n denotes a value of type X::native_handle_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 29: SocketAcceptorService requirements

expression	return type	assertion/note pre/post-condition
X::native_handle_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. post: !a.is_open(b).
a.destroy(b);		From IoObjectService requirements. Implicitly cancels asynchronous operations, as if by calling a.close(b, ec).
a.move_construct(b, c);		From IoObjectService requirements. The underlying native representation is moved from c to b.
a.move_assign(b, ao, c);		From IoObjectService requirements. Implicitly cancels asynchronous operations associated with b, as if by calling a.close(b, ec). Then the underlying native representation is moved from c to b.
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	

Table 29: (continued)

expression	return type	assertion/note pre/post-condition
a.close(b, ec);	error_code	If <code>a.is_open()</code> is true, causes any outstanding asynchronous operations to complete as soon as possible. Handlers for cancelled operations shall be passed the error code <code>error::operation_aborted</code> . post: <code>!a.is_open(b)</code> .
a.native_handle(b);	X::native_handle_type	
a.cancel(b, ec);	error_code	pre: <code>a.is_open(b)</code> . Causes any outstanding asynchronous operations to complete as soon as possible. Handlers for cancelled operations shall be passed the error code <code>error::operation_aborted</code> .
a.set_option(b, s, ec);	error_code	pre: <code>a.is_open(b)</code> .
a.get_option(b, g, ec);	error_code	pre: <code>a.is_open(b)</code> .
const X& u = a; const X::implementation_type& v = b; u.get_option(v, g, ec);	error_code	pre: <code>a.is_open(b)</code> .
a.io_control(b, i, ec);	error_code	pre: <code>a.is_open(b)</code> .
const typename Protocol::endpoint& u = e; a.bind(b, u, ec);	error_code	pre: <code>a.is_open(b)</code> .
a.local_endpoint(b, ec);	Protocol::endpoint	pre: <code>a.is_open(b)</code> .
const X& u = a; const X::implementation_type& v = b; u.local_endpoint(v, ec);	Protocol::endpoint	pre: <code>a.is_open(b)</code> .

Table 29: (continued)

expression	return type	assertion/note pre/post-condition
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.get_io_service() and behaves according to asynchronous operation 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.get_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.

5.42 Socket service requirements

A socket service must meet the requirements for an **I/O object service** with support for movability, as well as the additional requirements listed below.

In the table below, X denotes a socket service class for protocol **Protocol**, a and ao denote values of type X, b and c denote values of type X::implementation_type, p denotes a value of type Protocol, n denotes a value of type X::native_handle_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 30: SocketService requirements

expression	return type	assertion/note pre/post-condition
X::native_handle_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.move_construct(b, c);		From IoObjectService requirements. The underlying native representation is moved from c to b.
a.move_assign(b, ao, c);		From IoObjectService requirements. Implicitly cancels asynchronous operations associated with b, as if by calling a.close(b, ec). Then the underlying native representation is moved from c to b.
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 <code>error::operation_aborted</code> . post: !a.is_open(b).

Table 30: (continued)

expression	return type	assertion/note pre/post-condition
a.native_handle(b);	X::native_handle_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 <code>error::operation_aborted</code> .
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).

Table 30: (continued)

expression	return type	assertion/note pre/post-condition
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.get_io_service() and behaves according to asynchronous operation requirements .

5.43 Stream descriptor service requirements

A stream descriptor service must meet the requirements for a **descriptor service**, as well as the additional requirements listed below.

In the table below, X denotes a stream descriptor 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, 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 31: StreamDescriptorService requirements

expression	return type	assertion/note pre/post-condition
<code>a.read_some(b, mb, ec);</code>	<code>size_t</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Reads one or more bytes of data from a descriptor <code>b</code>.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>If successful, returns the number of bytes read. Otherwise returns 0. If the total size of all buffers in the sequence <code>mb</code> is 0, the function shall return 0 immediately.</p> <p>If the operation completes due to graceful connection closure by the peer, the operation shall fail with <code>error::eof</code>.</p>

Table 31: (continued)

expression	return type	assertion/note pre/post-condition
<pre>a.async_read_some(b, mb, rh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to read one or more bytes of data from a descriptor <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first. If the total size of all buffers in the sequence <code>mb</code> 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.</p> <p>If the operation completes due to graceful connection closure by the peer, the operation shall fail with <code>error::eof</code>.</p> <p>If the operation completes successfully, the <code>ReadHandler</code> object <code>rh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

Table 31: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.write_some(b, cb, ec);</code>	<code>size_t</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Writes one or more bytes of data to a descriptor <code>b</code>.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>If successful, returns the number of bytes written. Otherwise returns 0. If the total size of all buffers in the sequence <code>cb</code> is 0, the function shall return 0 immediately.</p>

Table 31: (continued)

expression	return type	assertion/note pre/post-condition
<pre>a.async_write_some(b, cb, wh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to write one or more bytes of data to a descriptor <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>The implementation shall maintain one or more copies of <code>cb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>cb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first. If the total size of all buffers in the sequence <code>cb</code> is 0, the asynchronous operation shall complete immediately and pass 0 as the argument to the handler that specifies the number of bytes read.</p> <p>If the operation completes successfully, the <code>WriteHandler</code> object <code>wh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

5.44 Stream handle service requirements

A stream handle service must meet the requirements for a **handle service**, as well as the additional requirements listed below.

In the table below, `X` denotes a stream handle 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`, `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 32: StreamHandleService requirements

expression	return type	assertion/note pre/post-condition
<code>a.read_some(b, mb, ec);</code>	<code>size_t</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Reads one or more bytes of data from a handle <code>b</code>.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>If successful, returns the number of bytes read. Otherwise returns 0. If the total size of all buffers in the sequence <code>mb</code> is 0, the function shall return 0 immediately.</p> <p>If the operation completes due to graceful connection closure by the peer, the operation shall fail with <code>error::eof</code>.</p>

Table 32: (continued)

expression	return type	assertion/note pre/post-condition
<pre>a.async_read_some(b, mb, rh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to read one or more bytes of data from a handle <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first. If the total size of all buffers in the sequence <code>mb</code> 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.</p> <p>If the operation completes due to graceful connection closure by the peer, the operation shall fail with <code>error::eof</code>.</p> <p>If the operation completes successfully, the <code>ReadHandler</code> object <code>rh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

Table 32: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.write_some(b, cb, ec);</code>	<code>size_t</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Writes one or more bytes of data to a handle <code>b</code>.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>If successful, returns the number of bytes written. Otherwise returns 0. If the total size of all buffers in the sequence <code>cb</code> is 0, the function shall return 0 immediately.</p>

Table 32: (continued)

expression	return type	assertion/note pre/post-condition
<pre>a.async_write_some(b, cb, wh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to write one or more bytes of data to a handle <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>The implementation shall maintain one or more copies of <code>cb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>cb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first. If the total size of all buffers in the sequence <code>cb</code> is 0, the asynchronous operation shall complete immediately and pass 0 as the argument to the handler that specifies the number of bytes read.</p> <p>If the operation completes successfully, the <code>WriteHandler</code> object <code>wh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

5.45 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 33: StreamSocketService requirements

expression	return type	assertion/note pre/post-condition
<code>a.receive(b, mb, f, ec);</code>	<code>size_t</code>	<p>pre: <code>a.is_open(b)</code>.</p> <p>Reads one or more bytes of data from a connected socket <code>b</code>.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>If successful, returns the number of bytes read. Otherwise returns 0. If the total size of all buffers in the sequence <code>mb</code> is 0, the function shall return 0 immediately.</p> <p>If the operation completes due to graceful connection closure by the peer, the operation shall fail with <code>error::eof</code>.</p>

Table 33: (continued)

expression	return type	assertion/note pre/post-condition
<pre>a.async_receive(b, mb, f, rh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to read one or more bytes of data from a connected socket <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>The implementation shall maintain one or more copies of <code>mb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>mb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first. If the total size of all buffers in the sequence <code>mb</code> 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.</p> <p>If the operation completes due to graceful connection closure by the peer, the operation shall fail with <code>error::eof</code>.</p> <p>If the operation completes successfully, the <code>ReadHandler</code> object <code>rh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

Table 33: (continued)

expression	return type	assertion/note pre/post-condition
a.send(b, cb, f, ec);	size_t	<p>pre: a.is_open(b).</p> <p>Writes one or more bytes of data to a connected socket b.</p> <p>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.</p> <p>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.</p>

Table 33: (continued)

expression	return type	assertion/note pre/post-condition
<pre>a.async_send(b, cb, f, wh);</pre>	void	<p>pre: <code>a.is_open(b)</code>.</p> <p>Initiates an asynchronous operation to write one or more bytes of data to a connected socket <code>b</code>. The operation is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements.</p> <p>The constant buffer sequence <code>cb</code> 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.</p> <p>The implementation shall maintain one or more copies of <code>cb</code> 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:</p> <ul style="list-style-type: none"> — the last copy of <code>cb</code> is destroyed, or — the handler for the asynchronous operation is invoked, <p>whichever comes first. If the total size of all buffers in the sequence <code>cb</code> is 0, the asynchronous operation shall complete immediately and pass 0 as the argument to the handler that specifies the number of bytes read.</p> <p>If the operation completes successfully, the <code>WriteHandler</code> object <code>wh</code> is invoked with the number of bytes transferred. Otherwise it is invoked with 0.</p>

5.46 Buffer-oriented synchronous random-access read device requirements

In the table below, `a` denotes a synchronous random-access read device object, `o` denotes an offset of type `boost::uint64_t`, `mb` denotes an object satisfying **mutable buffer sequence** requirements, and `ec` denotes an object of type `error_code`.

Table 34: Buffer-oriented synchronous random-access read device requirements

operation	type	semantics, pre/post-conditions
a.read_some_at(o, mb);	size_t	<pre>Equivalent to: error_code ec; size_t s = a.read_some_at(← o, mb, ec); if (ec) throw system_error ← (ec); return s;</pre>
a.read_some_at(o, mb, ec);	size_t	<p>Reads one or more bytes of data from the device <code>a</code> at offset <code>o</code>.</p> <p>The mutable buffer sequence <code>mb</code> specifies memory where the data should be placed. The <code>read_some_at</code> operation shall always fill a buffer in the sequence completely before proceeding to the next.</p> <p>If successful, returns the number of bytes read and sets <code>ec</code> such that <code>!ec</code> is true. If an error occurred, returns 0 and sets <code>ec</code> such that <code>!ec</code> is true.</p> <p>If the total size of all buffers in the sequence <code>mb</code> is 0, the function shall return 0 immediately.</p>

5.47 Buffer-oriented synchronous random-access write device requirements

In the table below, `a` denotes a synchronous random-access write device object, `o` denotes an offset of type `boost::uint64_t`, `cb` denotes an object satisfying **constant buffer sequence** requirements, and `ec` denotes an object of type `error_code`.

Table 35: Buffer-oriented synchronous random-access write device requirements

operation	type	semantics, pre/post-conditions
a.write_some_at(o, cb);	size_t	<pre>Equivalent to: error_code ec; size_t s = a.write_some(o, ← cb, ec); if (ec) throw system_error ← (ec); return s;</pre>

Table 35: (continued)

operation	type	semantics, pre/post-conditions
<code>a.write_some_at(o, cb, ec);</code>	<code>size_t</code>	<p>Writes one or more bytes of data to the device <code>a</code> at offset <code>o</code>.</p> <p>The constant buffer sequence <code>cb</code> specifies memory where the data to be written is located. The <code>write_some_at</code> operation shall always write a buffer in the sequence completely before proceeding to the next.</p> <p>If successful, returns the number of bytes written and sets <code>ec</code> such that <code>!ec</code> is true. If an error occurred, returns 0 and sets <code>ec</code> such that <code>!ec</code> is true.</p> <p>If the total size of all buffers in the sequence <code>cb</code> is 0, the function shall return 0 immediately.</p>

5.48 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 36: Buffer-oriented synchronous read stream requirements

operation	type	semantics, pre/post-conditions
<code>a.read_some(mb);</code>	<code>size_t</code>	<p>Equivalent to:</p> <pre>error_code ec; size_t s = a.read_some(mb, ← ec); if (ec) throw system_error ← (ec); return s;</pre>

Table 36: (continued)

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

5.49 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 37: Buffer-oriented synchronous write stream requirements

operation	type	semantics, pre/post-conditions
a.write_some(cb);	size_t	<p>Equivalent to:</p> <pre>error_code ec; size_t s = a.write_some(cb ← , ec); if (ec) throw system_error ← (ec); return s;</pre>

Table 37: (continued)

operation	type	semantics, pre/post-conditions
a.write_some(cb, ec);	size_t	<p>Writes one or more bytes of data to the stream a.</p> <p>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.</p> <p>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.</p> <p>If the total size of all buffers in the sequence cb is 0, the function shall return 0 immediately.</p>

5.50 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 38: 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.
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.

Table 38: (continued)

expression	return type	assertion/note pre/post-condition
X::less_than(t1, t2);	bool	Returns whether t1 is to be treated as less than t2.
X::to_posix_duration(d);	date_time::time_duration_type	Returns the date_time::time_duration_type value that most closely represents the duration d.

5.51 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 39: 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()).

Table 39: (continued)

expression	return type	assertion/note pre/post-condition
<code>a.expires_from_now(b, d, e ←);</code>	<code>size_t</code>	Equivalent to <code>a.expires_at(b, TimeTraits::add(TimeTraits::now(), d), e)</code> .
<code>a.wait(b, e);</code>	<code>error_code</code>	Sets <code>e</code> to indicate success or failure. Returns <code>e</code> . <code>post: !e !TimeTraits::lt(TimeTraits::now(), a.expires_at(b))</code> .
<code>a.async_wait(b, h);</code>		Initiates an asynchronous wait operation that is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements . The handler shall be posted for execution only if the condition <code>!ec !TimeTraits::lt(TimeTraits::now(), a.expires_at(b))</code> holds, where <code>ec</code> is the error code to be passed to the handler.

5.52 Waitable timer service requirements

A waitable 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 waitable timer service class for clock type `Clock`, where `Clock` meets the C++11 clock type requirements, `a` denotes a value of type `X`, `b` denotes a value of type `X::implementation_type`, `t` denotes a value of type `Clock::time_point`, `d` denotes a value of type `Clock::duration`, `e` denotes a value of type `error_code`, and `h` denotes a value meeting **WaitHandler** requirements.

Table 40: WaitableTimerService requirements

expression	return type	assertion/note pre/post-condition
<code>a.destroy(b);</code>		From IoObjectService requirements. Implicitly cancels asynchronous wait operations, as if by calling <code>a.cancel(b, e)</code> .

Table 40: (continued)

expression	return type	assertion/note pre/post-condition
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 <code>error::operation_aborted</code> . Sets <code>e</code> to indicate success or failure. Returns the number of operations that were cancelled.
a.expires_at(b);	Clock::time_point	
a.expires_at(b, t, e);	size_t	Implicitly cancels asynchronous wait operations, as if by calling <code>a.cancel(b, e)</code> . Returns the number of operations that were cancelled. post: <code>a.expires_at(b) == t</code> .
a.expires_from_now(b);	Clock::duration	Returns a value equivalent to <code>a.expires_at(b) - Clock::now()</code> .
a.expires_from_now(b, d, e ←);	size_t	Equivalent to <code>a.expires_at(b, Clock::now() + d, e)</code> .
a.wait(b, e);	error_code	Sets <code>e</code> to indicate success or failure. Returns <code>e</code> . post: <code>!!e !(Clock::now() < a.expires_at(b))</code> .
a.async_wait(b, h);		Initiates an asynchronous wait operation that is performed via the <code>io_service</code> object <code>a.get_io_service()</code> and behaves according to asynchronous operation requirements . The handler shall be posted for execution only if the condition <code>!!ec !(Clock::now() < a.expires_at(b))</code> holds, where <code>ec</code> is the error code to be passed to the handler.

5.53 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`.

Examples

A free function as a wait handler:

```
void wait_handler(
    const asio::error_code& ec)
{
    ...
}
```

A wait handler function object:

```
struct wait_handler
{
    ...
    void operator() (
        const asio::error_code& ec)
    {
        ...
    }
    ...
};
```

A non-static class member function adapted to a wait handler using bind():

```
void my_class::wait_handler(
    const asio::error_code& ec)
{
    ...
}
...
socket.async_wait(...,
    boost::bind(&my_class::wait_handler,
        this, asio::placeholders::error));
```

5.54 Wait traits requirements

In the table below, X denotes a wait traits class for clock type `Clock`, where `Clock` meets the C++11 type requirements for a clock, and `d` denotes a value of type `Clock::duration`.

Table 41: WaitTraits requirements

expression	return type	assertion/note pre/post-condition
<code>X::to_wait_duration(d);</code>	<code>Clock::duration</code>	Returns the maximum duration to be used for an individual, implementation-defined wait operation.

5.55 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`.

Examples

A free function as a write handler:

```
void write_handler(
    const asio::error_code& ec,
    std::size_t bytes_transferred)
{
    ...
}
```

A write handler function object:

```
struct write_handler
{
    ...
    void operator() (
        const asio::error_code& ec,
        std::size_t bytes_transferred)
    {
        ...
    }
    ...
};
```

A non-static class member function adapted to a write handler using bind():

```
void my_class::write_handler(
    const asio::error_code& ec,
    std::size_t bytes_transferred)
{
    ...
}
...
socket.async_write(...,
    boost::bind(&my_class::write_handler,
        this, asio::placeholders::error,
        asio::placeholders::bytes_transferred));
```

5.56 add_service

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

This function is used to add a service to the `io_service`.

Parameters

ios The `io_service` object that owns the service.

svc The service object. On success, ownership of the service object is transferred to the `io_service`. When the `io_service` object is destroyed, it will destroy the service object by performing:

```
delete static_cast<io_service::service*>(svc)
```

Exceptions

asio::service_already_exists Thrown if a service of the given type is already present in the **io_service**.

asio::invalid_service_owner Thrown if the service's owning **io_service** is not the **io_service** object specified by the **ios** parameter.

Requirements

Header: asio/io_service.hpp

Convenience header: asio.hpp

5.57 asio_handler_allocate

Default allocation function for handlers.

```
void * asio_handler_allocate(
    std::size_t size,
    ... );
```

Asynchronous operations may need to allocate temporary objects. Since asynchronous operations have a handler function object, these temporary objects can be said to be associated with the handler.

Implement `asio_handler_allocate` and `asio_handler_deallocate` for your own handlers to provide custom allocation for these temporary objects.

The default implementation of these allocation hooks uses `operator new` and `operator delete`.

Remarks

All temporary objects associated with a handler will be deallocated before the upcall to the handler is performed. This allows the same memory to be reused for a subsequent asynchronous operation initiated by the handler.

Example

```
class my_handler;

void* asio_handler_allocate(std::size_t size, my_handler* context)
{
    return ::operator new(size);
}

void asio_handler_deallocate(void* pointer, std::size_t size,
    my_handler* context)
{
    ::operator delete(pointer);
}
```

Requirements

Header: asio/handler_alloc_hook.hpp

Convenience header: asio.hpp

5.58 asio_handler_deallocate

Default deallocation function for handlers.

```
void asio_handler_deallocate(
    void * pointer,
    std::size_t size,
    ... );
```

Implement `asio_handler_allocate` and `asio_handler_deallocate` for your own handlers to provide custom allocation for the associated temporary objects.

The default implementation of these allocation hooks uses `operator new` and `operator delete`.

Requirements

Header: `asio/handler_alloc_hook.hpp`

Convenience header: `asio.hpp`

5.59 asio_handler_invoke

Default invoke function for handlers.

```
template<
    typename Function>
void asio_handler_invoke(
    Function & function,
    ... );

template<
    typename Function>
void asio_handler_invoke(
    const Function & function,
    ... );
```

Completion handlers for asynchronous operations are invoked by the `io_service` associated with the corresponding object (e.g. a socket or `deadline_timer`). Certain guarantees are made on when the handler may be invoked, in particular that a handler can only be invoked from a thread that is currently calling `run()` on the corresponding `io_service` object. Handlers may subsequently be invoked through other objects (such as `io_service::strand` objects) that provide additional guarantees.

When asynchronous operations are composed from other asynchronous operations, all intermediate handlers should be invoked using the same method as the final handler. This is required to ensure that user-defined objects are not accessed in a way that may violate the guarantees. This hooking function ensures that the invoked method used for the final handler is accessible at each intermediate step.

Implement `asio_handler_invoke` for your own handlers to specify a custom invocation strategy.

This default implementation invokes the function object like so:

```
function();
```

If necessary, the default implementation makes a copy of the function object so that the non-const `operator()` can be used.

Example

```
class my_handler;

template <typename Function>
void asio_handler_invoke(Function function, my_handler* context)
{
    context->strand_.dispatch(function);
}
```

Requirements

Header: asio/handler_invoke_hook.hpp

Convenience header: asio.hpp

5.59.1 asio_handler_invoke (1 of 2 overloads)

Default handler invocation hook used for non-const function objects.

```
template<
    typename Function>
void asio_handler_invoke(
    Function & function,
    ... );
```

5.59.2 asio_handler_invoke (2 of 2 overloads)

Default handler invocation hook used for const function objects.

```
template<
    typename Function>
void asio_handler_invoke(
    const Function & function,
    ... );
```

5.60 asio_handler_is_continuation

Default continuation function for handlers.

```
bool asio_handler_is_continuation(
    ... );
```

Asynchronous operations may represent a continuation of the asynchronous control flow associated with the current handler. The implementation can use this knowledge to optimise scheduling of the handler.

Implement asio_handler_is_continuation for your own handlers to indicate when a handler represents a continuation.

The default implementation of the continuation hook returns `false`.

Example

```
class my_handler;

bool asio_handler_is_continuation(my_handler* context)
{
    return true;
}
```

Requirements

Header: asio/handler_continuation_hook.hpp

Convenience header: asio.hpp

5.61 `async_connect`

Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ComposedConnectHandler>
void-or-deduced async_connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    ComposedConnectHandler handler);

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ComposedConnectHandler>
void-or-deduced async_connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    Iterator end,
    ComposedConnectHandler handler);

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition,
    typename ComposedConnectHandler>
void-or-deduced async_connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    ConnectCondition connect_condition,
    ComposedConnectHandler handler);

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition,
    typename ComposedConnectHandler>
void-or-deduced async_connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    Iterator end,
    ConnectCondition connect_condition,
    ComposedConnectHandler handler);
```

Requirements

Header: asio/connect.hpp

Convenience header: asio.hpp

5.61.1 `async_connect` (1 of 4 overloads)

Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ComposedConnectHandler>
void-or-deduced async_connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    ComposedConnectHandler handler);
```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `async_connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

s The socket to be connected. If the socket is already open, it will be closed.

begin An iterator pointing to the start of a sequence of endpoints.

handler The handler to be called when the connect operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    // Result of operation. if the sequence is empty, set to
    // asio::error::not_found. Otherwise, contains the
    // error from the last connection attempt.
    const asio::error_code& error,

    // On success, an iterator denoting the successfully
    // connected endpoint. Otherwise, the end iterator.
    Iterator iterator
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

This overload assumes that a default constructed object of type `Iterator` represents the end of the sequence. This is a valid assumption for iterator types such as `asio::ip::tcp::resolver::iterator`.

Example

```

tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::socket s(io_service);

// ...

r.async_resolve(q, resolve_handler);

// ...

void resolve_handler(
    const asio::error_code& ec,
    tcp::resolver::iterator i)
{
    if (!ec)
    {
        asio::async_connect(s, i, connect_handler);
    }
}

// ...

void connect_handler(
    const asio::error_code& ec,
    tcp::resolver::iterator i)
{
    // ...
}

```

5.61.2 `async_connect` (2 of 4 overloads)

Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ComposedConnectHandler>
void-or-deduced async_connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    Iterator end,
    ComposedConnectHandler handler);

```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `async_connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

s The socket to be connected. If the socket is already open, it will be closed.

begin An iterator pointing to the start of a sequence of endpoints.

end An iterator pointing to the end of a sequence of endpoints.

handler The handler to be called when the connect operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    // Result of operation. if the sequence is empty, set to
    // asio::error::not_found. Otherwise, contains the
    // error from the last connection attempt.
    const asio::error_code& error,

    // On success, an iterator denoting the successfully
    // connected endpoint. Otherwise, the end iterator.
    Iterator iterator
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

```

tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::socket s(io_service);

// ...

r.async_resolve(q, resolve_handler);

// ...

void resolve_handler(
    const asio::error_code& ec,
    tcp::resolver::iterator i)
{
    if (!ec)
    {
        tcp::resolver::iterator end;
        asio::async_connect(s, i, end, connect_handler);
    }
}

// ...

void connect_handler(
    const asio::error_code& ec,
    tcp::resolver::iterator i)
{
    // ...
}

```

5.61.3 `async_connect` (3 of 4 overloads)

Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition,
    typename ComposedConnectHandler>
void-or-deduced async_connect(
    basic_socket< Protocol, SocketService > & s,

```

```
Iterator begin,
ConnectCondition connect_condition,
ComposedConnectHandler handler);
```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `async_connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

s The socket to be connected. If the socket is already open, it will be closed.

begin An iterator pointing to the start of a sequence of endpoints.

connect_condition A function object that is called prior to each connection attempt. The signature of the function object must be:

```
Iterator connect_condition(
    const asio::error_code& ec,
    Iterator next);
```

The `ec` parameter contains the result from the most recent connect operation. Before the first connection attempt, `ec` is always set to indicate success. The `next` parameter is an iterator pointing to the next endpoint to be tried. The function object should return the next iterator, but is permitted to return a different iterator so that endpoints may be skipped. The implementation guarantees that the function object will never be called with the end iterator.

handler The handler to be called when the connect operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    // Result of operation. if the sequence is empty, set to
    // asio::error::not_found. Otherwise, contains the
    // error from the last connection attempt.
    const asio::error_code& error,

    // On success, an iterator denoting the successfully
    // connected endpoint. Otherwise, the end iterator.
    Iterator iterator
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

This overload assumes that a default constructed object of type `Iterator` represents the end of the sequence. This is a valid assumption for iterator types such as `asio::ip::tcp::resolver::iterator`.

Example

The following connect condition function object can be used to output information about the individual connection attempts:

```
struct my_connect_condition
{
    template <typename Iterator>
    Iterator operator() (
        const asio::error_code& ec,
        Iterator next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next->endpoint() << std::endl;
    }
};
```

```

        return next;
    }
};

```

It would be used with the `asio::connect` function as follows:

```

tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::socket s(io_service);

// ...

r.async_resolve(q, resolve_handler);

// ...

void resolve_handler(
    const asio::error_code& ec,
    tcp::resolver::iterator i)
{
if (!ec)
{
    asio::async_connect(s, i,
        my_connect_condition(),
        connect_handler);
}
}

// ...

void connect_handler(
    const asio::error_code& ec,
    tcp::resolver::iterator i)
{
if (ec)
{
    // An error occurred.
}
else
{
    std::cout << "Connected to: " << i->endpoint() << std::endl;
}
}

```

5.61.4 `async_connect` (4 of 4 overloads)

Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition,
    typename ComposedConnectHandler>
void-or-deduced async_connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    Iterator end,
    ConnectCondition connect_condition,
    ComposedConnectHandler handler);

```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `async_connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

s The socket to be connected. If the socket is already open, it will be closed.

begin An iterator pointing to the start of a sequence of endpoints.

end An iterator pointing to the end of a sequence of endpoints.

connect_condition A function object that is called prior to each connection attempt. The signature of the function object must be:

```
Iterator connect_condition(
    const asio::error_code& ec,
    Iterator next);
```

The `ec` parameter contains the result from the most recent connect operation. Before the first connection attempt, `ec` is always set to indicate success. The `next` parameter is an iterator pointing to the next endpoint to be tried. The function object should return the next iterator, but is permitted to return a different iterator so that endpoints may be skipped. The implementation guarantees that the function object will never be called with the end iterator.

handler The handler to be called when the connect operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    // Result of operation. if the sequence is empty, set to
    // asio::error::not_found. Otherwise, contains the
    // error from the last connection attempt.
    const asio::error_code& error,

    // On success, an iterator denoting the successfully
    // connected endpoint. Otherwise, the end iterator.
    Iterator iterator
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

The following connect condition function object can be used to output information about the individual connection attempts:

```
struct my_connect_condition
{
    template <typename Iterator>
    Iterator operator()(const asio::error_code& ec, Iterator next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next->endpoint() << std::endl;
        return next;
    }
};
```

It would be used with the `asio::connect` function as follows:

```

tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::socket s(io_service);

// ...

r.async_resolve(q, resolve_handler);

// ...

void resolve_handler(
    const asio::error_code& ec,
    tcp::resolver::iterator i)
{
    if (!ec)
    {
        tcp::resolver::iterator end;
        asio::async_connect(s, i, end,
            my_connect_condition(),
            connect_handler);
    }
}

// ...

void connect_handler(
    const asio::error_code& ec,
    tcp::resolver::iterator i)
{
    if (ec)
    {
        // An error occurred.
    }
    else
    {
        std::cout << "Connected to: " << i->endpoint() << std::endl;
    }
}

```

5.62 async_read

Start an asynchronous operation to read a certain amount of data from a stream.

```

template<
    typename AsyncReadStream,
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read(
    AsyncReadStream & s,
    const MutableBufferSequence & buffers,
    ReadHandler handler);

template<
    typename AsyncReadStream,
    typename MutableBufferSequence,
    typename CompletionCondition,
    typename ReadHandler>
void-or-deduced async_read(

```

```

AsyncReadStream & s,
const MutableBufferSequence & buffers,
CompletionCondition completion_condition,
ReadHandler handler);

template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
void-or-deduced async_read(
    AsyncReadStream & s,
    basic_streambuf< Allocator > & b,
    ReadHandler handler);

template<
    typename AsyncReadStream,
    typename Allocator,
    typename CompletionCondition,
    typename ReadHandler>
void-or-deduced async_read(
    AsyncReadStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    ReadHandler handler);

```

Requirements

Header: asio/read.hpp

Convenience header: asio.hpp

5.62.1 `async_read` (1 of 4 overloads)

Start an asynchronous operation to read a certain amount of data from a stream.

```

template<
    typename AsyncReadStream,
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read(
    AsyncReadStream & s,
    const MutableBufferSequence & buffers,
    ReadHandler handler);

```

This function is used to asynchronously read a certain number of bytes of data from a stream. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `async_read_some` function, and is known as a *composed operation*. The program must ensure that the stream performs no other read operations (such as `async_read`, the stream's `async_read_some` function, or any other composed operations that perform reads) until this operation completes.

Parameters

s The stream from which the data is to be read. The type must support the `AsyncReadStream` concept.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the stream. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.

    std::size_t bytes_transferred           // Number of bytes copied into the
                                              // buffers. If an error occurred,
                                              // this will be the number of
                                              // bytes successfully transferred
                                              // prior to the error.

);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To read into a single data buffer use the `buffer` function as follows:

```
asio::async_read(s, asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

Remarks

This overload is equivalent to calling:

```
asio::async_read(
    s, buffers,
    asio::transfer_all(),
    handler);
```

5.62.2 `async_read` (2 of 4 overloads)

Start an asynchronous operation to read a certain amount of data from a stream.

```
template<
    typename AsyncReadStream,
    typename MutableBufferSequence,
    typename CompletionCondition,
    typename ReadHandler>
void-or-deduced async_read(
    AsyncReadStream & s,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    ReadHandler handler);
```

This function is used to asynchronously read a certain number of bytes of data from a stream. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The completion_condition function object returns 0.

Parameters

s The stream from which the data is to be read. The type must support the `AsyncReadStream` concept.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the stream. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest async_read_some operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the stream's `async_read_some` function.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.

    std::size_t bytes_transferred           // Number of bytes copied into the
                                            // buffers. If an error occurred,
                                            // this will be the number of
                                            // bytes successfully transferred
                                            // prior to the error.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To read into a single data buffer use the **buffer** function as follows:

```
asio::async_read(s,
    asio::buffer(data, size),
    asio::transfer_at_least(32),
    handler);
```

See the **buffer** documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.62.3 `async_read` (3 of 4 overloads)

Start an asynchronous operation to read a certain amount of data from a stream.

```
template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
void-or-deduced async_read(
    AsyncReadStream & s,
    basic_streambuf< Allocator > & b,
    ReadHandler handler);
```

This function is used to asynchronously read a certain number of bytes of data from a stream. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- The supplied buffer is full (that is, it has reached maximum size).
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `async_read_some` function, and is known as a *composed operation*. The program must ensure that the stream performs no other read operations (such as `async_read`, the stream's `async_read_some` function, or any other composed operations that perform reads) until this operation completes.

Parameters

s The stream from which the data is to be read. The type must support the `AsyncReadStream` concept.

b A `basic_streambuf` object into which the data will be read. Ownership of the `streambuf` is retained by the caller, which must guarantee that it remains valid until the handler is called.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.

    std::size_t bytes_transferred           // Number of bytes copied into the
                                              // buffers. If an error occurred,
                                              // this will be the number of
                                              // bytes successfully transferred
                                              // prior to the error.

);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

This overload is equivalent to calling:

```
asio::async_read(
    s, b,
    asio::transfer_all(),
    handler);
```

5.62.4 `async_read` (4 of 4 overloads)

Start an asynchronous operation to read a certain amount of data from a stream.

```
template<
    typename AsyncReadStream,
    typename Allocator,
    typename CompletionCondition,
    typename ReadHandler>
void-or-deduced async_read(
    AsyncReadStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    ReadHandler handler);
```

This function is used to asynchronously read a certain number of bytes of data from a stream. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- The supplied buffer is full (that is, it has reached maximum size).
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's `async_read_some` function, and is known as a *composed operation*. The program must ensure that the stream performs no other read operations (such as `async_read`, the stream's `async_read_some` function, or any other composed operations that perform reads) until this operation completes.

Parameters

- s** The stream from which the data is to be read. The type must support the `AsyncReadStream` concept.
- b** A `basic_streambuf` object into which the data will be read. Ownership of the `streambuf` is retained by the caller, which must guarantee that it remains valid until the handler is called.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest async_read_some operation.
    const asio::error_code& error,

    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the stream's `async_read_some` function.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.

    std::size_t bytes_transferred           // Number of bytes copied into the
                                              // buffers. If an error occurred,
                                              // this will be the number of
                                              // bytes successfully transferred
                                              // prior to the error.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.63 `async_read_at`

Start an asynchronous operation to read a certain amount of data at the specified offset.

```
template<
    typename AsyncRandomAccessReadDevice,
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    ReadHandler handler);

template<
    typename AsyncRandomAccessReadDevice,
    typename MutableBufferSequence,
    typename CompletionCondition,
    typename ReadHandler>
void-or-deduced async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    ReadHandler handler);

template<
    typename AsyncRandomAccessReadDevice,
    typename Allocator,
    typename ReadHandler>
void-or-deduced async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    ReadHandler handler);

template<
    typename AsyncRandomAccessReadDevice,
    typename Allocator,
    typename CompletionCondition,
    typename ReadHandler>
void-or-deduced async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    ReadHandler handler);
```

Requirements

Header: `asio/read_at.hpp`

Convenience header: `asio.hpp`

5.63.1 `async_read_at` (1 of 4 overloads)

Start an asynchronous operation to read a certain amount of data at the specified offset.

```

template<
    typename AsyncRandomAccessReadDevice,
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    ReadHandler handler);

```

This function is used to asynchronously read a certain number of bytes of data from a random access device at the specified offset. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the device's `async_read_some_at` function.

Parameters

d The device from which the data is to be read. The type must support the `AsyncRandomAccessReadDevice` concept.

offset The offset at which the data will be read.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the device. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    // Result of operation.
    const asio::error_code& error,

    // Number of bytes copied into the buffers. If an error
    // occurred, this will be the number of bytes successfully
    // transferred prior to the error.
    std::size_t bytes_transferred
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To read into a single data buffer use the `buffer` function as follows:

```
asio::async_read_at(d, 42, asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

Remarks

This overload is equivalent to calling:

```
asio::async_read_at(
    d, 42, buffers,
    asio::transfer_all(),
    handler);
```

5.63.2 `async_read_at` (2 of 4 overloads)

Start an asynchronous operation to read a certain amount of data at the specified offset.

```
template<
    typename AsyncRandomAccessReadDevice,
    typename MutableBufferSequence,
    typename CompletionCondition,
    typename ReadHandler>
void-or-deduced async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    ReadHandler handler);
```

This function is used to asynchronously read a certain number of bytes of data from a random access device at the specified offset. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The `completion_condition` function object returns 0.

Parameters

d The device from which the data is to be read. The type must support the `AsyncRandomAccessReadDevice` concept.

offset The offset at which the data will be read.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the device. Although the `buffers` object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest async_read_some_at operation.
    const asio::error_code& error,

    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the device's `async_read_some_at` function.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    // Result of operation.
    const asio::error_code& error,
    // Number of bytes copied into the buffers. If an error
    // occurred, this will be the number of bytes successfully
    // transferred prior to the error.
    std::size_t bytes_transferred
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To read into a single data buffer use the `buffer` function as follows:

```

asio::async_read_at(d, 42,
    asio::buffer(data, size),
    asio::transfer_at_least(32),
    handler);

```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.63.3 `async_read_at` (3 of 4 overloads)

Start an asynchronous operation to read a certain amount of data at the specified offset.

```

template<
    typename AsyncRandomAccessReadDevice,
    typename Allocator,
    typename ReadHandler>
void-or-deduced async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    ReadHandler handler);

```

This function is used to asynchronously read a certain number of bytes of data from a random access device at the specified offset. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- An error occurred.

This operation is implemented in terms of zero or more calls to the device's `async_read_some_at` function.

Parameters

d The device from which the data is to be read. The type must support the `AsyncRandomAccessReadDevice` concept.

offset The offset at which the data will be read.

b A `basic_streambuf` object into which the data will be read. Ownership of the streambuf is retained by the caller, which must guarantee that it remains valid until the handler is called.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    // Result of operation.
    const asio::error_code& error,
    // Number of bytes copied into the buffers. If an error
    // occurred, this will be the number of bytes successfully
    // transferred prior to the error.
    std::size_t bytes_transferred
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

This overload is equivalent to calling:

```

asio::async_read_at(
    d, 42, b,
    asio::transfer_all(),
    handler);

```

5.63.4 `async_read_at` (4 of 4 overloads)

Start an asynchronous operation to read a certain amount of data at the specified offset.

```

template<
    typename AsyncRandomAccessReadDevice,
    typename Allocator,
    typename CompletionCondition,
    typename ReadHandler>
void-or-deduced async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    ReadHandler handler);

```

This function is used to asynchronously read a certain number of bytes of data from a random access device at the specified offset. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the device's `async_read_some_at` function.

Parameters

d The device from which the data is to be read. The type must support the `AsyncRandomAccessReadDevice` concept.

offset The offset at which the data will be read.

b A `basic_streambuf` object into which the data will be read. Ownership of the streambuf is retained by the caller, which must guarantee that it remains valid until the handler is called.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```

std::size_t completion_condition(
    // Result of latest async_read_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);

```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the device's `async_read_some_at` function.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    // Result of operation.
    const asio::error_code& error,
    // Number of bytes copied into the buffers. If an error
    // occurred, this will be the number of bytes successfully
    // transferred prior to the error.
    std::size_t bytes_transferred
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.64 `async_read_until`

Start an asynchronous operation to read data into a `streambuf` until it contains a delimiter, matches a regular expression, or a function object indicates a match.

```

template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
void-or-deduced async_read_until(
    AsyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    char delim,
    ReadHandler handler);

template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
void-or-deduced async_read_until(
    AsyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    const std::string & delim,
    ReadHandler handler);

template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
void-or-deduced async_read_until(
    AsyncReadStream & s,

```

```

asio::basic_streambuf< Allocator > & b,
const boost::regex & expr,
ReadHandler handler);

template<
    typename AsyncReadStream,
    typename Allocator,
    typename MatchCondition,
    typename ReadHandler>
void-or-deduced async_read_until(
    AsyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    MatchCondition match_condition,
    ReadHandler handler,
    typename enable_if< is_match_condition< MatchCondition >::value >::type * = 0);

```

Requirements

Header: asio/read_until.hpp

Convenience header: asio.hpp

5.64.1 `async_read_until` (1 of 4 overloads)

Start an asynchronous operation to read data into a streambuf until it contains a specified delimiter.

```

template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
void-or-deduced async_read_until(
    AsyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    char delim,
    ReadHandler handler);

```

This function is used to asynchronously read data into the specified streambuf until the streambuf's get area contains the specified delimiter. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- The get area of the streambuf contains the specified delimiter.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `async_read_some` function, and is known as a *composed operation*. If the streambuf's get area already contains the delimiter, this asynchronous operation completes immediately. The program must ensure that the stream performs no other read operations (such as `async_read`, `async_read_until`, the stream's `async_read_some` function, or any other composed operations that perform reads) until this operation completes.

Parameters

- s** The stream from which the data is to be read. The type must support the `AsyncReadStream` concept.
- b** A streambuf object into which the data will be read. Ownership of the streambuf is retained by the caller, which must guarantee that it remains valid until the handler is called.

delim The delimiter character.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    // Result of operation.
    const asio::error_code& error,
    // The number of bytes in the streambuf's get
    // area up to and including the delimiter.
    // 0 if an error occurred.
    std::size_t bytes_transferred
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

After a successful `async_read_until` operation, the streambuf may contain additional data beyond the delimiter. An application will typically leave that data in the streambuf for a subsequent `async_read_until` operation to examine.

Example

To asynchronously read data into a streambuf until a newline is encountered:

```
asio::streambuf b;
...
void handler(const asio::error_code& e, std::size_t size)
{
    if (!e)
    {
        std::istream is(&b);
        std::string line;
        std::getline(is, line);
        ...
    }
}
...
asio::async_read_until(s, b, '\n', handler);
```

After the `async_read_until` operation completes successfully, the buffer `b` contains the delimiter:

```
{ 'a', 'b', ..., 'c', '\n', 'd', 'e', ... }
```

The call to `std::getline` then extracts the data up to and including the delimiter, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\n' }
```

The remaining data is left in the buffer `b` as follows:

```
{ 'd', 'e', ... }
```

This data may be the start of a new line, to be extracted by a subsequent `async_read_until` operation.

5.64.2 `async_read_until` (2 of 4 overloads)

Start an asynchronous operation to read data into a `streambuf` until it contains a specified delimiter.

```
template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
void-or-deduced async_read_until(
    AsyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    const std::string & delim,
    ReadHandler handler);
```

This function is used to asynchronously read data into the specified `streambuf` until the `streambuf`'s get area contains the specified delimiter. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- The get area of the `streambuf` contains the specified delimiter.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `async_read_some` function, and is known as a *composed operation*. If the `streambuf`'s get area already contains the delimiter, this asynchronous operation completes immediately. The program must ensure that the stream performs no other read operations (such as `async_read`, `async_read_until`, the stream's `async_read_some` function, or any other composed operations that perform reads) until this operation completes.

Parameters

s The stream from which the data is to be read. The type must support the `AsyncReadStream` concept.

b A `streambuf` object into which the data will be read. Ownership of the `streambuf` is retained by the caller, which must guarantee that it remains valid until the handler is called.

delim The delimiter string.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    // Result of operation.
    const asio::error_code& error,
    // The number of bytes in the streambuf's get
    // area up to and including the delimiter.
    // 0 if an error occurred.
    std::size_t bytes_transferred
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

After a successful `async_read_until` operation, the `streambuf` may contain additional data beyond the delimiter. An application will typically leave that data in the `streambuf` for a subsequent `async_read_until` operation to examine.

Example

To asynchronously read data into a streambuf until a newline is encountered:

```
asio::streambuf b;
...
void handler(const asio::error_code& e, std::size_t size)
{
    if (!e)
    {
        std::istream is(&b);
        std::string line;
        std::getline(is, line);
        ...
    }
}
...
asio::async_read_until(s, b, "\r\n", handler);
```

After the `async_read_until` operation completes successfully, the buffer `b` contains the delimiter:

```
{ 'a', 'b', ..., 'c', '\r', '\n', 'd', 'e', ... }
```

The call to `std::getline` then extracts the data up to and including the delimiter, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r', '\n' }
```

The remaining data is left in the buffer `b` as follows:

```
{ 'd', 'e', ... }
```

This data may be the start of a new line, to be extracted by a subsequent `async_read_until` operation.

5.64.3 `async_read_until` (3 of 4 overloads)

Start an asynchronous operation to read data into a streambuf until some part of its data matches a regular expression.

```
template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
void-or-deduced async_read_until(
    AsyncReadStream & s,
    asio::basic_streambuf<Allocator> & b,
    const boost::regex & expr,
    ReadHandler handler);
```

This function is used to asynchronously read data into the specified streambuf until the streambuf's get area contains some data that matches a regular expression. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- A substring of the streambuf's get area matches the regular expression.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `async_read_some` function, and is known as a *composed operation*. If the streambuf's get area already contains data that matches the regular expression, this asynchronous operation completes immediately. The program must ensure that the stream performs no other read operations (such as `async_read`, `async_read_until`, the stream's `async_read_some` function, or any other composed operations that perform reads) until this operation completes.

Parameters

- s** The stream from which the data is to be read. The type must support the `AsyncReadStream` concept.
- b** A `streambuf` object into which the data will be read. Ownership of the `streambuf` is retained by the caller, which must guarantee that it remains valid until the handler is called.
- expr** The regular expression.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(  
    // Result of operation.  
    const asio::error_code& error,  
  
    // The number of bytes in the streambuf's get  
    // area up to and including the substring  
    // that matches the regular expression.  
    // 0 if an error occurred.  
    std::size_t bytes_transferred  
) ;
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

After a successful `async_read_until` operation, the `streambuf` may contain additional data beyond that which matched the regular expression. An application will typically leave that data in the `streambuf` for a subsequent `async_read_until` operation to examine.

Example

To asynchronously read data into a `streambuf` until a CR-LF sequence is encountered:

```
asio::streambuf b;  
...  
void handler(const asio::error_code& e, std::size_t size)  
{  
    if (!e)  
    {  
        std::istream is(&b);  
        std::string line;  
        std::getline(is, line);  
        ...  
    }  
    ...  
    asio::async_read_until(s, b, boost::regex("\r\n"), handler);  
}
```

After the `async_read_until` operation completes successfully, the buffer `b` contains the data which matched the regular expression:

```
{ 'a', 'b', ..., 'c', '\r', '\n', 'd', 'e', ... }
```

The call to `std::getline` then extracts the data up to and including the match, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r', '\n' }
```

The remaining data is left in the buffer `b` as follows:

```
{ 'd', 'e', ... }
```

This data may be the start of a new line, to be extracted by a subsequent `async_read_until` operation.

5.64.4 `async_read_until` (4 of 4 overloads)

Start an asynchronous operation to read data into a `streambuf` until a function object indicates a match.

```
template<
    typename AsyncReadStream,
    typename Allocator,
    typename MatchCondition,
    typename ReadHandler>
void-or-deduced async_read_until(
    AsyncReadStream & s,
    asio::basic_streambuf<Allocator> & b,
    MatchCondition match_condition,
    ReadHandler handler,
    typename enable_if< is_match_condition<MatchCondition>::value >::type * = 0);
```

This function is used to asynchronously read data into the specified `streambuf` until a user-defined match condition function object, when applied to the data contained in the `streambuf`, indicates a successful match. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- The match condition function object returns a `std::pair` where the second element evaluates to true.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `async_read_some` function, and is known as a *composed operation*. If the match condition function object already indicates a match, this asynchronous operation completes immediately. The program must ensure that the stream performs no other read operations (such as `async_read`, `async_read_until`, the stream's `async_read_some` function, or any other composed operations that perform reads) until this operation completes.

Parameters

s The stream from which the data is to be read. The type must support the `AsyncReadStream` concept.

b A `streambuf` object into which the data will be read.

match_condition The function object to be called to determine whether a match exists. The signature of the function object must be:

```
pair<iterator, bool> match_condition(iterator begin, iterator end);
```

where `iterator` represents the type:

```
buffers_iterator<basic_streambuf<Allocator>>::const_buffers_type>
```

The iterator parameters `begin` and `end` define the range of bytes to be scanned to determine whether there is a match. The first member of the return value is an iterator marking one-past-the-end of the bytes that have been consumed by the match function. This iterator is used to calculate the `begin` parameter for any subsequent invocation of the match condition. The second member of the return value is true if a match has been found, false otherwise.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    // Result of operation.
    const asio::error_code& error,
    // The number of bytes in the streambuf's get
    // area that have been fully consumed by the
    // match function. 0 if an error occurred.
    std::size_t bytes_transferred
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

After a successful `async_read_until` operation, the streambuf may contain additional data beyond that which matched the function object. An application will typically leave that data in the streambuf for a subsequent `async_read_until` operation to examine.

The default implementation of the `is_match_condition` type trait evaluates to true for function pointers and function objects with a `result_type` typedef. It must be specialised for other user-defined function objects.

Examples

To asynchronously read data into a streambuf until whitespace is encountered:

```

typedef asio::buffers_iterator<
    asio::streambuf::const_buffers_type> iterator;

std::pair<iterator, bool>
match_whitespace(iterator begin, iterator end)
{
    iterator i = begin;
    while (i != end)
        if (std::isspace(*i++))
            return std::make_pair(i, true);
    return std::make_pair(i, false);
}
...
void handler(const asio::error_code& e, std::size_t size);
...
asio::streambuf b;
asio::async_read_until(s, b, match_whitespace, handler);

```

To asynchronously read data into a streambuf until a matching character is found:

```

class match_char
{
public:
    explicit match_char(char c) : c_(c) {}

    template <typename Iterator>
    std::pair<Iterator, bool> operator()(Iterator begin, Iterator end) const
    {
        Iterator i = begin;
        while (i != end)
            if (c_ == *i++)
                return std::make_pair(i, true);
        return std::make_pair(i, false);
    }
}

```

```

}

private:
    char c_;
};

namespaceasio{
    template<>structis_match_condition<match_char>
        : public boost::true_type {};
} // namespaceasio
...
void handler(constasio::error_code&e, std::size_tsize);
...
asio::streambufb;
asio::async_read_until(s,b,match_char('a'),handler);

```

5.65 `async_result`

An interface for customising the behaviour of an initiating function.

```

template<
    typename Handler>
class async_result

```

Types

Name	Description
type	The return type of the initiating function.

Member Functions

Name	Description
async_result	Construct an async result from a given handler.
get	Obtain the value to be returned from the initiating function.

This template may be specialised for user-defined handler types.

Requirements

Header: `asio/async_result.hpp`

Convenience header: `asio.hpp`

5.65.1 `async_result::async_result`

Construct an async result from a given handler.

```
async_result(
    Handler & );
```

When using a specialised `async_result`, the constructor has an opportunity to initialise some state associated with the handler, which is then returned from the initiating function.

5.65.2 `async_result::get`

Obtain the value to be returned from the initiating function.

```
type get();
```

5.65.3 `async_result::type`

The return type of the initiating function.

```
typedef void type;
```

Requirements

Header: `asio/async_result.hpp`

Convenience header: `asio.hpp`

5.66 `async_write`

Start an asynchronous operation to write a certain amount of data to a stream.

```
template<
    typename AsyncWriteStream,
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write(
    AsyncWriteStream & s,
    const ConstBufferSequence & buffers,
    WriteHandler handler);

template<
    typename AsyncWriteStream,
    typename ConstBufferSequence,
    typename CompletionCondition,
    typename WriteHandler>
void-or-deduced async_write(
    AsyncWriteStream & s,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    WriteHandler handler);

template<
    typename AsyncWriteStream,
    typename Allocator,
    typename WriteHandler>
void-or-deduced async_write(
    AsyncWriteStream & s,
```

```

basic_streambuf< Allocator > & b,
WriteHandler handler);

template<
    typename AsyncWriteStream,
    typename Allocator,
    typename CompletionCondition,
    typename WriteHandler>
void-or-deduced async_write(
    AsyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    WriteHandler handler);

```

Requirements

Header: asio/write.hpp

Convenience header: asio.hpp

5.66.1 `async_write` (1 of 4 overloads)

Start an asynchronous operation to write all of the supplied data to a stream.

```

template<
    typename AsyncWriteStream,
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write(
    AsyncWriteStream & s,
    const ConstBufferSequence & buffers,
    WriteHandler handler);

```

This function is used to asynchronously write a certain number of bytes of data to a stream. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `async_write_some` function, and is known as a *composed operation*. The program must ensure that the stream performs no other write operations (such as `async_write`, the stream's `async_write_some` function, or any other composed operations that perform writes) until this operation completes.

Parameters

s The stream to which the data is to be written. The type must support the `AsyncWriteStream` concept.

buffers One or more buffers containing the data to be written. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    const asio::error_code& error, // Result of operation.

    std::size_t bytes_transferred           // Number of bytes written from the
                                            // buffers. If an error occurred,
                                            // this will be less than the sum
                                            // of the buffer sizes.
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To write a single data buffer use the `buffer` function as follows:

```
asio::async_write(s, asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.66.2 `async_write` (2 of 4 overloads)

Start an asynchronous operation to write a certain amount of data to a stream.

```

template<
    typename AsyncWriteStream,
    typename ConstBufferSequence,
    typename CompletionCondition,
    typename WriteHandler>
void-or-deduced async_write(
    AsyncWriteStream & s,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    WriteHandler handler);

```

This function is used to asynchronously write a certain number of bytes of data to a stream. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's `async_write_some` function, and is known as a *composed operation*. The program must ensure that the stream performs no other write operations (such as `async_write`, the stream's `async_write_some` function, or any other composed operations that perform writes) until this operation completes.

Parameters

s The stream to which the data is to be written. The type must support the `AsyncWriteStream` concept.

buffers One or more buffers containing the data to be written. Although the `buffers` object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

completion_condition The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```

    std::size_t completion_condition(
        // Result of latest async_write_some operation.
        const asio::error_code& error,
        // Number of bytes transferred so far.
        std::size_t bytes_transferred
    );

```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the stream's `async_write_some` function.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    const asio::error_code& error, // Result of operation.

    std::size_t bytes_transferred           // Number of bytes written from the
                                              // buffers. If an error occurred,
                                              // this will be less than the sum
                                              // of the buffer sizes.
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To write a single data buffer use the `buffer` function as follows:

```

asio::async_write(s,
    asio::buffer(data, size),
    asio::transfer_at_least(32),
    handler);

```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.66.3 `async_write` (3 of 4 overloads)

Start an asynchronous operation to write all of the supplied data to a stream.

```

template<
    typename AsyncWriteStream,
    typename Allocator,
    typename WriteHandler>
void-or-deduced async_write(
    AsyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    WriteHandler handler);

```

This function is used to asynchronously write a certain number of bytes of data to a stream. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- All of the data in the supplied `basic_streambuf` has been written.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `async_write_some` function, and is known as a *composed operation*. The program must ensure that the stream performs no other write operations (such as `async_write`, the stream's `async_write_some` function, or any other composed operations that perform writes) until this operation completes.

Parameters

- s** The stream to which the data is to be written. The type must support the `AsyncWriteStream` concept.
 - b** A `basic_streambuf` object from which data will be written. Ownership of the `streambuf` is retained by the caller, which must guarantee that it remains valid until the handler is called.
- handler** The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.

    std::size_t bytes_transferred           // Number of bytes written from the
                                              // buffers. If an error occurred,
                                              // this will be less than the sum
                                              // of the buffer sizes.

);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.66.4 `async_write` (4 of 4 overloads)

Start an asynchronous operation to write a certain amount of data to a stream.

```
template<
    typename AsyncWriteStream,
    typename Allocator,
    typename CompletionCondition,
    typename WriteHandler>
void-or-deduced async_write(
    AsyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    WriteHandler handler);
```

This function is used to asynchronously write a certain number of bytes of data to a stream. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- All of the data in the supplied `basic_streambuf` has been written.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's `async_write_some` function, and is known as a *composed operation*. The program must ensure that the stream performs no other write operations (such as `async_write`, the stream's `async_write_some` function, or any other composed operations that perform writes) until this operation completes.

Parameters

- s** The stream to which the data is to be written. The type must support the `AsyncWriteStream` concept.
 - b** A `basic_streambuf` object from which data will be written. Ownership of the `streambuf` is retained by the caller, which must guarantee that it remains valid until the handler is called.
- completion_condition** The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```

    std::size_t completion_condition(
        // Result of latest async_write_some operation.
        const asio::error_code& error,
        // Number of bytes transferred so far.
        std::size_t bytes_transferred
    );
}

```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the stream's `async_write_some` function.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    const asio::error_code& error, // Result of operation.

    std::size_t bytes_transferred // Number of bytes written from the
                                // buffers. If an error occurred,
                                // this will be less than the sum
                                // of the buffer sizes.
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.67 `async_write_at`

Start an asynchronous operation to write a certain amount of data at the specified offset.

```

template<
    typename AsyncRandomAccessWriteDevice,
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_at(  

    AsyncRandomAccessWriteDevice & d,  

    uint64_t offset,  

    const ConstBufferSequence & buffers,  

    WriteHandler handler);

template<
    typename AsyncRandomAccessWriteDevice,
    typename ConstBufferSequence,
    typename CompletionCondition,
    typename WriteHandler>
void-or-deduced async_write_at(  

    AsyncRandomAccessWriteDevice & d,  

    uint64_t offset,  

    const ConstBufferSequence & buffers,  

    CompletionCondition completion_condition,  

    WriteHandler handler);

template<
    typename AsyncRandomAccessWriteDevice,
    typename Allocator,
    typename WriteHandler>
void-or-deduced async_write_at(  

    AsyncRandomAccessWriteDevice & d,

```

```

    uint64_t offset,
    basic_streambuf< Allocator > & b,
    WriteHandler handler);

template<
    typename AsyncRandomAccessWriteDevice,
    typename Allocator,
    typename CompletionCondition,
    typename WriteHandler>
void-or-deduced async_write_at(
    AsyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    WriteHandler handler);

```

Requirements

Header: asio/write_at.hpp

Convenience header: asio.hpp

5.67.1 `async_write_at` (1 of 4 overloads)

Start an asynchronous operation to write all of the supplied data at the specified offset.

```

template<
    typename AsyncRandomAccessWriteDevice,
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_at(
    AsyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    WriteHandler handler);

```

This function is used to asynchronously write a certain number of bytes of data to a random access device at a specified offset. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the device's `async_write_some_at` function, and is known as a *composed operation*. The program must ensure that the device performs no *overlapping* write operations (such as `async_write_at`, the device's `async_write_some_at` function, or any other composed operations that perform writes) until this operation completes. Operations are overlapping if the regions defined by their offsets, and the numbers of bytes to write, intersect.

Parameters

d The device to which the data is to be written. The type must support the `AsyncRandomAccessWriteDevice` concept.

offset The offset at which the data will be written.

buffers One or more buffers containing the data to be written. Although the `buffers` object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    // Result of operation.
    const asio::error_code& error,
    // Number of bytes written from the buffers. If an error
    // occurred, this will be less than the sum of the buffer sizes.
    std::size_t bytes_transferred
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To write a single data buffer use the `buffer` function as follows:

```
asio::async_write_at(d, 42, asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.67.2 `async_write_at` (2 of 4 overloads)

Start an asynchronous operation to write a certain amount of data at the specified offset.

```
template<
    typename AsyncRandomAccessWriteDevice,
    typename ConstBufferSequence,
    typename CompletionCondition,
    typename WriteHandler>
void-or-deduced async_write_at(
    AsyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    WriteHandler handler);
```

This function is used to asynchronously write a certain number of bytes of data to a random access device at a specified offset. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the device's `async_write_some_at` function, and is known as a *composed operation*. The program must ensure that the device performs no *overlapping* write operations (such as `async_write_at`, the device's `async_write_some_at` function, or any other composed operations that perform writes) until this operation completes. Operations are overlapping if the regions defined by their offsets, and the numbers of bytes to write, intersect.

Parameters

d The device to which the data is to be written. The type must support the `AsyncRandomAccessWriteDevice` concept.

offset The offset at which the data will be written.

buffers One or more buffers containing the data to be written. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

completion_condition The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest async_write_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the device's `async_write_some_at` function.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    // Result of operation.
    const asio::error_code& error,
    // Number of bytes written from the buffers. If an error
    // occurred, this will be less than the sum of the buffer sizes.
    std::size_t bytes_transferred
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To write a single data buffer use the `buffer` function as follows:

```
asio::async_write_at(d, 42,
    asio::buffer(data, size),
    asio::transfer_at_least(32),
    handler);
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.67.3 `async_write_at` (3 of 4 overloads)

Start an asynchronous operation to write all of the supplied data at the specified offset.

```
template<
    typename AsyncRandomAccessWriteDevice,
    typename Allocator,
    typename WriteHandler>
void-or-deduced async_write_at(
    AsyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    WriteHandler handler);
```

This function is used to asynchronously write a certain number of bytes of data to a random access device at a specified offset. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- All of the data in the supplied `basic_streambuf` has been written.
- An error occurred.

This operation is implemented in terms of zero or more calls to the device's `async_write_some_at` function, and is known as a *composed operation*. The program must ensure that the device performs no *overlapping* write operations (such as `async_write_at`, the device's `async_write_some_at` function, or any other composed operations that perform writes) until this operation completes. Operations are overlapping if the regions defined by their offsets, and the numbers of bytes to write, intersect.

Parameters

- d** The device to which the data is to be written. The type must support the `AsyncRandomAccessWriteDevice` concept.
- offset** The offset at which the data will be written.
- b** A `basic_streambuf` object from which data will be written. Ownership of the `streambuf` is retained by the caller, which must guarantee that it remains valid until the handler is called.
- handler** The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    // Result of operation.
    const asio::error_code& error,
    // Number of bytes written from the buffers. If an error
    // occurred, this will be less than the sum of the buffer sizes.
    std::size_t bytes_transferred
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.67.4 `async_write_at` (4 of 4 overloads)

Start an asynchronous operation to write a certain amount of data at the specified offset.

```
template<
    typename AsyncRandomAccessWriteDevice,
    typename Allocator,
    typename CompletionCondition,
    typename WriteHandler>
void-or-deduced async_write_at(
    AsyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    WriteHandler handler);
```

This function is used to asynchronously write a certain number of bytes of data to a random access device at a specified offset. The function call always returns immediately. The asynchronous operation will continue until one of the following conditions is true:

- All of the data in the supplied `basic_streambuf` has been written.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the device's `async_write_some_at` function, and is known as a *composed operation*. The program must ensure that the device performs no *overlapping* write operations (such as `async_write_at`, the device's `async_write_some_at` function, or any other composed operations that perform writes) until this operation completes. Operations are overlapping if the regions defined by their offsets, and the numbers of bytes to write, intersect.

Parameters

- d The device to which the data is to be written. The type must support the `AsyncRandomAccessWriteDevice` concept.
 - offset The offset at which the data will be written.
 - b A `basic_streambuf` object from which data will be written. Ownership of the `streambuf` is retained by the caller, which must guarantee that it remains valid until the handler is called.
- completion_condition** The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest async_write_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the device's `async_write_some_at` function.

- handler** The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    // Result of operation.
    const asio::error_code& error,
    // Number of bytes written from the buffers. If an error
    // occurred, this will be less than the sum of the buffer sizes.
    std::size_t bytes_transferred
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.68 basic_datagram_socket

Provides datagram-oriented socket functionality.

```
template<
    typename Protocol,
    typename DatagramSocketService = datagram_socket_service<Protocol>>
class basic_datagram_socket :
    public basic_socket< Protocol, DatagramSocketService >
```

Types

Name	Description
<code>broadcast</code>	Socket option to permit sending of broadcast messages.
<code>bytes_readable</code>	IO control command to get the amount of data that can be read without blocking.

Name	Description
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.

Name	Description
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive on a connected socket.
async_receive_from	Start an asynchronous receive.
async_send	Start an asynchronous send on a connected socket.
async_send_to	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_datagram_socket	Construct a basic_datagram_socket without opening it. Construct and open a basic_datagram_socket. Construct a basic_datagram_socket, opening it and binding it to the given local endpoint. Construct a basic_datagram_socket on an existing native socket. Move-construct a basic_datagram_socket from another. Move-construct a basic_datagram_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.

Name	Description
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.
operator=	Move-assign a basic_datagram_socket from another. Move-assign a basic_datagram_socket from a socket of another protocol type.
receive	Receive some data on a connected socket.
receive_from	Receive a datagram with the endpoint of the sender.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on a connected socket.
send_to	Send a datagram to the specified endpoint.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.

Name	Description
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_datagram_socket` class template provides asynchronous and blocking datagram-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/basic_datagram_socket.hpp`

Convenience header: `asio.hpp`

5.68.1 basic_datagram_socket::assign

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);

asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.68.1.1 basic_datagram_socket::assign (1 of 2 overloads)

Inherited from basic_socket.

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

5.68.1.2 basic_datagram_socket::assign (2 of 2 overloads)

Inherited from `basic_socket`.

Assign an existing native socket to the socket.

```
asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.68.2 basic_datagram_socket::async_connect

Inherited from `basic_socket`.

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
void or deduced async_connect(
    const endpoint_type & peer_endpoint,
    ConnectHandler handler);
```

This function is used to asynchronously connect a socket to the specified remote endpoint. The function call always returns immediately.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected. Copies will be made of the endpoint object as required.

handler The handler to be called when the connection operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error // Result of operation
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

```
void connect_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Connect succeeded.
    }
}

...

asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.async_connect(endpoint, connect_handler);
```

5.68.3 basic_datagram_socket::async_receive

Start an asynchronous receive on a connected socket.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    ReadHandler handler);

template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    ReadHandler handler);
```

5.68.3.1 basic_datagram_socket::async_receive (1 of 2 overloads)

Start an asynchronous receive on a connected socket.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

This function is used to asynchronously receive data from the datagram socket. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The `async_receive` operation can only be used with a connected socket. Use the `async_receive_from` function to receive data on an unconnected datagram socket.

Example

To receive into a single data buffer use the **buffer** function as follows:

```
socket.async_receive(asio::buffer(data, size), handler);
```

See the **buffer** documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

5.68.3.2 basic_datagram_socket::async_receive (2 of 2 overloads)

Start an asynchronous receive on a connected socket.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    ReadHandler handler);
```

This function is used to asynchronously receive data from the datagram socket. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

flags Flags specifying how the receive call is to be made.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The `async_receive` operation can only be used with a connected socket. Use the `async_receive_from` function to receive data on an unconnected datagram socket.

5.68.4 basic_datagram_socket::async_receive_from

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    ReadHandler handler);
```

```

template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    ReadHandler handler);

```

5.68.4.1 basic_datagram_socket::async_receive_from (1 of 2 overloads)

Start an asynchronous receive.

```

template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    ReadHandler handler);

```

This function is used to asynchronously receive a datagram. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

sender_endpoint An endpoint object that receives the endpoint of the remote sender of the datagram. Ownership of the sender_endpoint object is retained by the caller, which must guarantee that it is valid until the handler is called.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To receive into a single data buffer use the **buffer** function as follows:

```

socket.async_receive_from(
    asio::buffer(data, size), sender_endpoint, handler);

```

See the **buffer** documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.68.4.2 basic_datagram_socket::async_receive_from (2 of 2 overloads)

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    ReadHandler handler);
```

This function is used to asynchronously receive a datagram. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

sender_endpoint An endpoint object that receives the endpoint of the remote sender of the datagram. Ownership of the sender_endpoint object is retained by the caller, which must guarantee that it is valid until the handler is called.

flags Flags specifying how the receive call is to be made.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.68.5 basic_datagram_socket::async_send

Start an asynchronous send on a connected socket.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    WriteHandler handler);

template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);
```

5.68.5.1 basic_datagram_socket::async_send (1 of 2 overloads)

Start an asynchronous send on a connected socket.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

This function is used to asynchronously send data on the datagram socket. The function call always returns immediately.

Parameters

buffers One or more data buffers to be sent on the socket. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes sent.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The `async_send` operation can only be used with a connected socket. Use the `async_send_to` function to send data on an unconnected datagram socket.

Example

To send a single data buffer use the `buffer` function as follows:

```
socket.async_send(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.68.5.2 basic_datagram_socket::async_send (2 of 2 overloads)

Start an asynchronous send on a connected socket.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);
```

This function is used to asynchronously send data on the datagram socket. The function call always returns immediately.

Parameters

buffers One or more data buffers to be sent on the socket. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

flags Flags specifying how the send call is to be made.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes sent.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The `async_send` operation can only be used with a connected socket. Use the `async_send_to` function to send data on an unconnected datagram socket.

5.68.6 basic_datagram_socket::async_send_to

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    WriteHandler handler);

template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    WriteHandler handler);
```

5.68.6.1 basic_datagram_socket::async_send_to (1 of 2 overloads)

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    WriteHandler handler);
```

This function is used to asynchronously send a datagram to the specified remote endpoint. The function call always returns immediately.

Parameters

buffers One or more data buffers to be sent to the remote endpoint. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

destination The remote endpoint to which the data will be sent. Copies will be made of the endpoint as required.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes sent.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To send a single data buffer use the `buffer` function as follows:

```
asio::ip::udp::endpoint destination(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.async_send_to(
    asio::buffer(data, size), destination, handler);
```

See the `buffer` documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.68.6.2 basic_datagram_socket::async_send_to (2 of 2 overloads)

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    WriteHandler handler);
```

This function is used to asynchronously send a datagram to the specified remote endpoint. The function call always returns immediately.

Parameters

buffers One or more data buffers to be sent to the remote endpoint. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

flags Flags specifying how the send call is to be made.

destination The remote endpoint to which the data will be sent. Copies will be made of the endpoint as required.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes sent.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.68.7 `basic_datagram_socket::at_mark`

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;

bool at_mark(
    asio::error_code & ec) const;
```

5.68.7.1 `basic_datagram_socket::at_mark (1 of 2 overloads)`

Inherited from basic_socket.

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

Exceptions

`asio::system_error` Thrown on failure.

5.68.7.2 `basic_datagram_socket::at_mark (2 of 2 overloads)`

Inherited from basic_socket.

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark(
    asio::error_code & ec) const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

5.68.8 basic_datagram_socket::available

Determine the number of bytes available for reading.

```
std::size_t available() const;  
  
std::size_t available(  
    asio::error_code & ec) const;
```

5.68.8.1 basic_datagram_socket::available (1 of 2 overloads)

Inherited from basic_socket.

Determine the number of bytes available for reading.

```
std::size_t available() const;
```

This function is used to determine the number of bytes that may be read without blocking.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

Exceptions

asio::system_error Thrown on failure.

5.68.8.2 basic_datagram_socket::available (2 of 2 overloads)

Inherited from basic_socket.

Determine the number of bytes available for reading.

```
std::size_t available(  
    asio::error_code & ec) const;
```

This function is used to determine the number of bytes that may be read without blocking.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

5.68.9 basic_datagram_socket::basic_datagram_socket

Construct a `basic_datagram_socket` without opening it.

```
explicit basic_datagram_socket(
    asio::io_service & io_service);
```

Construct and open a `basic_datagram_socket`.

```
basic_datagram_socket(
    asio::io_service & io_service,
    const protocol_type & protocol);
```

Construct a `basic_datagram_socket`, opening it and binding it to the given local endpoint.

```
basic_datagram_socket(
    asio::io_service & io_service,
    const endpoint_type & endpoint);
```

Construct a `basic_datagram_socket` on an existing native socket.

```
basic_datagram_socket(
    asio::io_service & io_service,
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

Move-construct a `basic_datagram_socket` from another.

```
basic_datagram_socket(
    basic_datagram_socket && other);
```

Move-construct a `basic_datagram_socket` from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename DatagramSocketService1>
basic_datagram_socket(
    basic_datagram_socket< Protocol1, DatagramSocketService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.68.9.1 basic_datagram_socket::basic_datagram_socket (1 of 6 overloads)

Construct a `basic_datagram_socket` without opening it.

```
basic_datagram_socket(
    asio::io_service & io_service);
```

This constructor creates a datagram socket without opening it. The `open()` function must be called before data can be sent or received on the socket.

Parameters

io_service The `io_service` object that the datagram socket will use to dispatch handlers for any asynchronous operations performed on the socket.

5.68.9.2 basic_datagram_socket::basic_datagram_socket (2 of 6 overloads)

Construct and open a [basic_datagram_socket](#).

```
basic_datagram_socket(
    asio::io_service & io_service,
    const protocol_type & protocol);
```

This constructor creates and opens a datagram socket.

Parameters

io_service The [io_service](#) object that the datagram socket will use to dispatch handlers for any asynchronous operations performed on the socket.

protocol An object specifying protocol parameters to be used.

Exceptions

[asio::system_error](#) Thrown on failure.

5.68.9.3 basic_datagram_socket::basic_datagram_socket (3 of 6 overloads)

Construct a [basic_datagram_socket](#), opening it and binding it to the given local endpoint.

```
basic_datagram_socket(
    asio::io_service & io_service,
    const endpoint_type & endpoint);
```

This constructor creates a datagram socket and automatically opens it bound to the specified endpoint on the local machine. The protocol used is the protocol associated with the given endpoint.

Parameters

io_service The [io_service](#) object that the datagram socket will use to dispatch handlers for any asynchronous operations performed on the socket.

endpoint An endpoint on the local machine to which the datagram socket will be bound.

Exceptions

[asio::system_error](#) Thrown on failure.

5.68.9.4 basic_datagram_socket::basic_datagram_socket (4 of 6 overloads)

Construct a [basic_datagram_socket](#) on an existing native socket.

```
basic_datagram_socket(
    asio::io_service & io_service,
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

This constructor creates a datagram socket object to hold an existing native socket.

Parameters

io_service The `io_service` object that the datagram socket will use to dispatch handlers for any asynchronous operations performed on the socket.

protocol An object specifying protocol parameters to be used.

native_socket The new underlying socket implementation.

Exceptions

`asio::system_error` Thrown on failure.

5.68.9.5 `basic_datagram_socket::basic_datagram_socket (5 of 6 overloads)`

Move-construct a `basic_datagram_socket` from another.

```
basic_datagram_socket(
    basic_datagram_socket && other);
```

This constructor moves a datagram socket from one object to another.

Parameters

other The other `basic_datagram_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_datagram_socket(io_service&)` constructor.

5.68.9.6 `basic_datagram_socket::basic_datagram_socket (6 of 6 overloads)`

Move-construct a `basic_datagram_socket` from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename DatagramSocketService1>
basic_datagram_socket(
    basic_datagram_socket< Protocol1, DatagramSocketService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

This constructor moves a datagram socket from one object to another.

Parameters

other The other `basic_datagram_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_datagram_socket(io_service&)` constructor.

5.68.10 basic_datagram_socket::bind

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.68.10.1 basic_datagram_socket::bind (1 of 2 overloads)

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345));
```

5.68.10.2 basic_datagram_socket::bind (2 of 2 overloads)

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
asio::error_code ec;
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345), ec);
if (ec)
{
    // An error occurred.
}
```

5.68.11 basic_datagram_socket::broadcast

Inherited from socket_base.

Socket option to permit sending of broadcast messages.

```
typedef implementation_defined broadcast;
```

Implements the SOL_SOCKET/SO_BROADCAST socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.12 basic_datagram_socket::bytes_readable

Inherited from socket_base.

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::bytes_readable command(true);
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.13 basic_datagram_socket::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.68.13.1 basic_datagram_socket::cancel (1 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
void cancel();
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.68.13.2 basic_datagram_socket::cancel (2 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.68.14 basic_datagram_socket::close

Close the socket.

```
void close();

asio::error_code close(
    asio::error_code & ec);
```

5.68.14.1 basic_datagram_socket::close (1 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
void close();
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure. Note that, even if the function indicates an error, the underlying descriptor is closed.

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.68.14.2 `basic_datagram_socket::close` (2 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
asio::error_code close(
    asio::error_code & ec);
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any. Note that, even if the function indicates an error, the underlying descriptor is closed.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
socket.close(ec);
if (ec)
{
    // An error occurred.
}
```

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.68.15 `basic_datagram_socket::connect`

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);

asio::error_code connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.68.15.1 basic_datagram_socket::connect (1 of 2 overloads)

Inherited from basic_socket.

Connect the socket to the specified endpoint.

```
void connect(  
    const endpoint_type & peer_endpoint);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);  
asio::ip::tcp::endpoint endpoint(  
    asio::ip::address::from_string("1.2.3.4"), 12345);  
socket.connect(endpoint);
```

5.68.15.2 basic_datagram_socket::connect (2 of 2 overloads)

Inherited from basic_socket.

Connect the socket to the specified endpoint.

```
asio::error_code connect(  
    const endpoint_type & peer_endpoint,  
    asio::error_code & ec);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
asio::error_code ec;
socket.connect(endpoint, ec);
if (ec)
{
    // An error occurred.
}
```

5.68.16 basic_datagram_socket::debug

Inherited from socket_base.

Socket option to enable socket-level debugging.

```
typedef implementation_defined debug;
```

Implements the SOL_SOCKET/SO_DEBUG socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.17 basic_datagram_socket::do_not_route

Inherited from socket_base.

Socket option to prevent routing, use local interfaces only.

```
typedef implementation_defined do_not_route;
```

Implements the SOL_SOCKET/SO_DONTROUTE socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.18 basic_datagram_socket::enable_connection_aborted

Inherited from socket_base.

Socket option to report aborted connections on accept.

```
typedef implementation_defined enable_connection_aborted;
```

Implements a custom socket option that determines whether or not an accept operation is permitted to fail with asio::error::connection_aborted. By default the option is false.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.19 basic_datagram_socket::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.20 basic_datagram_socket::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.68.20.1 basic_datagram_socket::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.68.20.2 basic_datagram_socket::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.68.21 basic_datagram_socket::get_io_service

Inherited from basic_io_object.

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the **io_service** object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the **io_service** object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.68.22 basic_datagram_socket::get_option

Get an option from the socket.

```
void get_option(
    GettableSocketOption & option) const;

asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.68.22.1 basic_datagram_socket::get_option (1 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.68.22.2 basic_datagram_socket::get_option (2 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
asio::error_code ec;
socket.get_option(option, ec);
if (ec)
{
    // An error occurred.
}
bool is_set = option.value();
```

5.68.23 basic_datagram_socket::get_service

Get the service associated with the I/O object.

```
service_type & get_service();

const service_type & get_service() const;
```

5.68.23.1 basic_datagram_socket::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.68.23.2 basic_datagram_socket::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.68.24 basic_datagram_socket::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.68.25 basic_datagram_socket::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.26 basic_datagram_socket::io_control

Perform an IO control command on the socket.

```
void io_control(
    IoControlCommand & command);

asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.68.26.1 basic_datagram_socket::io_control (1 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

5.68.26.2 basic_datagram_socket::io_control (2 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
asio::error_code ec;
socket.io_control(command, ec);
if (ec)
{
    // An error occurred.
}
std::size_t bytes_readable = command.get();
```

5.68.27 basic_datagram_socket::is_open

Inherited from basic_socket.

Determine whether the socket is open.

```
bool is_open() const;
```

5.68.28 basic_datagram_socket::keep_alive

Inherited from socket_base.

Socket option to send keep-alives.

```
typedef implementation_defined keep_alive;
```

Implements the SOL_SOCKET/SO_KEEPALIVE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.29 basic_datagram_socket::linger

Inherited from socket_base.

Socket option to specify whether the socket lingers on close if unsent data is present.

```
typedef implementation_defined linger;
```

Implements the SOL_SOCKET/SO_LINGER socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option;
socket.get_option(option);
bool is_set = option.enabled();
unsigned short timeout = option.timeout();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.30 basic_datagram_socket::local_endpoint

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;  
  
endpoint_type local_endpoint(  
    asio::error_code & ec) const;
```

5.68.30.1 basic_datagram_socket::local_endpoint (1 of 2 overloads)

Inherited from basic_socket.

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;
```

This function is used to obtain the locally bound endpoint of the socket.

Return Value

An object that represents the local endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);  
...  
asio::ip::tcp::endpoint endpoint = socket.local_endpoint();
```

5.68.30.2 basic_datagram_socket::local_endpoint (2 of 2 overloads)

Inherited from basic_socket.

Get the local endpoint of the socket.

```
endpoint_type local_endpoint(  
    asio::error_code & ec) const;
```

This function is used to obtain the locally bound endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the local endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.local_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.68.31 `basic_datagram_socket::lowest_layer`

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.68.31.1 `basic_datagram_socket::lowest_layer (1 of 2 overloads)`

Inherited from basic_socket.

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a `basic_socket` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.68.31.2 `basic_datagram_socket::lowest_layer (2 of 2 overloads)`

Inherited from basic_socket.

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a `basic_socket` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.68.32 basic_datagram_socket::lowest_layer_type

Inherited from `basic_socket`.

A `basic_socket` is always the lowest layer.

```
typedef basic_socket< Protocol, DatagramSocketService > lowest_layer_type;
```

Types

Name	Description
<code>broadcast</code>	Socket option to permit sending of broadcast messages.
<code>bytes_readable</code>	IO control command to get the amount of data that can be read without blocking.
<code>debug</code>	Socket option to enable socket-level debugging.
<code>do_not_route</code>	Socket option to prevent routing, use local interfaces only.
<code>enable_connection_aborted</code>	Socket option to report aborted connections on accept.
<code>endpoint_type</code>	The endpoint type.
<code>implementation_type</code>	The underlying implementation type of I/O object.
<code>keep_alive</code>	Socket option to send keep-alives.
<code>linger</code>	Socket option to specify whether the socket lingers on close if unsent data is present.
<code>lowest_layer_type</code>	A <code>basic_socket</code> is always the lowest layer.
<code>message_flags</code>	Bitmask type for flags that can be passed to send and receive operations.
<code>native_handle_type</code>	The native representation of a socket.
<code>native_type</code>	(Deprecated: Use <code>native_handle_type</code> .) The native representation of a socket.
<code>non_blocking_io</code>	(Deprecated: Use <code>non_blocking()</code> .) IO control command to set the blocking mode of the socket.
<code>protocol_type</code>	The protocol type.
<code>receive_buffer_size</code>	Socket option for the receive buffer size of a socket.
<code>receive_low_watermark</code>	Socket option for the receive low watermark.
<code>reuse_address</code>	Socket option to allow the socket to be bound to an address that is already in use.
<code>send_buffer_size</code>	Socket option for the send buffer size of a socket.

Name	Description
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_socket	Construct a basic_socket without opening it. Construct and open a basic_socket. Construct a basic_socket, opening it and binding it to the given local endpoint. Construct a basic_socket on an existing native socket. Move-construct a basic_socket from another. Move-construct a basic_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.

Name	Description
native_handle	Get the native socket representation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.
operator=	Move-assign a basic_socket from another. Move-assign a basic_socket from a socket of another protocol type.
remote_endpoint	Get the remote endpoint of the socket.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_socket	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_socket` class template provides functionality that is common to both stream-oriented and datagram-oriented sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/basic_datagram_socket.hpp`

Convenience header: `asio.hpp`

5.68.33 basic_datagram_socket::max_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.68.34 basic_datagram_socket::message_do_not_route

Inherited from socket_base.

Specify that the data should not be subject to routing.

```
static const int message_do_not_route = implementation_defined;
```

5.68.35 basic_datagram_socket::message_end_of_record

Inherited from socket_base.

Specifies that the data marks the end of a record.

```
static const int message_end_of_record = implementation_defined;
```

5.68.36 basic_datagram_socket::message_flags

Inherited from socket_base.

Bitmask type for flags that can be passed to send and receive operations.

```
typedef int message_flags;
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.37 basic_datagram_socket::message_out_of_band

Inherited from socket_base.

Process out-of-band data.

```
static const int message_out_of_band = implementation_defined;
```

5.68.38 basic_datagram_socket::message_peek

Inherited from socket_base.

Peek at incoming data without removing it from the input queue.

```
static const int message_peek = implementation_defined;
```

5.68.39 basic_datagram_socket::native

Inherited from basic_socket.

(Deprecated: Use native_handle().) Get the native socket representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.68.40 basic_datagram_socket::native_handle

Inherited from basic_socket.

Get the native socket representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.68.41 basic_datagram_socket::native_handle_type

The native representation of a socket.

```
typedef DatagramSocketService::native_handle_type native_handle_type;
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.42 basic_datagram_socket::native_non_blocking

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.68.42.1 basic_datagram_socket::native_non_blocking (1 of 3 overloads)

Inherited from basic_socket.

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

This function is used to retrieve the non-blocking mode of the underlying native socket. This mode has no effect on the behaviour of the socket object's synchronous operations.

Return Value

true if the underlying socket is in non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Remarks

The current non-blocking mode is cached by the socket object. Consequently, the return value may be incorrect if the non-blocking mode was set directly on the native socket.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
```

```

// Put the underlying socket into non-blocking mode.
if (!ec)
    if (!sock_.native_non_blocking())
        sock_.native_non_blocking(true, ec);

if (!ec)
{
    for (;;)
    {
        // Try the system call.
        errno = 0;
        int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
        ec = asio::error_code(n < 0 ? errno : 0,
            asio::error::get_system_category());
        total_bytes_transferred_ += ec ? 0 : n;

        // Retry operation immediately if interrupted by signal.
        if (ec == asio::error::interrupted)
            continue;

        // Check if we need to run the operation again.
        if (ec == asio::error::would_block
            || ec == asio::error::try_again)
        {
            // We have to wait for the socket to become ready again.
            sock_.async_write_some(asio::null_buffers(), *this);
            return;
        }

        if (ec || n == 0)
        {
            // An error occurred, or we have reached the end of the file.
            // Either way we must exit the loop so we can call the handler.
            break;
        }

        // Loop around to try calling sendfile again.
    }
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}

};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.68.42.2 basic_datagram_socket::native_non_blocking (2 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);
```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Exceptions

`asio::system_error` Thrown on failure. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.
                errno = 0;
                int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
                ec = asio::error_code(n < 0 ? errno : 0,
                                      asio::error::get_system_category());
                total_bytes_transferred_ += ec ? 0 : n;

                // Retry operation immediately if interrupted by signal.
                if (ec == asio::error::interrupted)
                    continue;

                // Check if we need to run the operation again.
                if (ec == asio::error::would_block
                    || ec == asio::error::try_again)
                {
                    // We have to wait for the socket to become ready again.
                    sock_.async_write_some(asio::null_buffers(), *this);
                }
            }
        }
    }
};
```

```

        return;
    }

    if (ec || n == 0)
    {
        // An error occurred, or we have reached the end of the file.
        // Either way we must exit the loop so we can call the handler.
        break;
    }

    // Loop around to try calling sendfile again.
}
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}
};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.68.42.3 basic_datagram_socket::native_non_blocking (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```
asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

ec Set to indicate what error occurred, if any. If the mode is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
```

```

off_t offset_;
std::size_t total_bytes_transferred_;

// Function call operator meeting WriteHandler requirements.
// Used as the handler for the async_write_some operation.
void operator()(asio::error_code ec, std::size_t)
{
    // Put the underlying socket into non-blocking mode.
    if (!ec)
        if (!sock_.native_non_blocking())
            sock_.native_non_blocking(true, ec);

    if (!ec)
    {
        for (;;)
        {
            // Try the system call.
            errno = 0;
            int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
            ec = asio::error_code(n < 0 ? errno : 0,
                asio::error::get_system_category());
            total_bytes_transferred_ += ec ? 0 : n;

            // Retry operation immediately if interrupted by signal.
            if (ec == asio::error::interrupted)
                continue;

            // Check if we need to run the operation again.
            if (ec == asio::error::would_block
                || ec == asio::error::try_again)
            {
                // We have to wait for the socket to become ready again.
                sock_.async_write_some(asio::null_buffers(), *this);
                return;
            }

            if (ec || n == 0)
            {
                // An error occurred, or we have reached the end of the file.
                // Either way we must exit the loop so we can call the handler.
                break;
            }

            // Loop around to try calling sendfile again.
        }
    }

    // Pass result back to user's handler.
    handler_(ec, total_bytes_transferred_);
}

};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.68.43 basic_datagram_socket::native_type

(Deprecated: Use native_handle_type.) The native representation of a socket.

```
typedef DatagramSocketService::native_handle_type native_type;
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.44 basic_datagram_socket::non_blocking

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Sets the non-blocking mode of the socket.

```
void non_blocking(
    bool mode);

asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.68.44.1 basic_datagram_socket::non_blocking (1 of 3 overloads)

Inherited from basic_socket.

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Return Value

true if the socket's synchronous operations will fail with asio::error::would_block if they are unable to perform the requested operation immediately. If false, synchronous operations will block until complete.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error asio::error::would_block.

5.68.44.2 basic_datagram_socket::non_blocking (2 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
void non_blocking(
    bool mode);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.68.44.3 `basic_datagram_socket::non_blocking` (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

ec Set to indicate what error occurred, if any.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.68.45 `basic_datagram_socket::non_blocking_io`

Inherited from socket_base.

(Deprecated: Use `non_blocking()`) IO control command to set the blocking mode of the socket.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::non_blocking_io command(true);
socket.io_control(command);
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.46 basic_datagram_socket::open

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());  
  
asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.68.46.1 basic_datagram_socket::open (1 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying protocol parameters to be used.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
```

5.68.46.2 basic_datagram_socket::open (2 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying which protocol is to be used.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.68.47 basic_datagram_socket::operator=

Move-assign a **basic_datagram_socket** from another.

```
basic_datagram_socket & operator=
    (basic_datagram_socket && other);
```

Move-assign a **basic_datagram_socket** from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename DatagramSocketService1>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_datagram_socket >::type & ←
operator=(←
    basic_datagram_socket< Protocol1, DatagramSocketService1 > && other);
```

5.68.47.1 basic_datagram_socket::operator= (1 of 2 overloads)

Move-assign a **basic_datagram_socket** from another.

```
basic_datagram_socket & operator=
    (basic_datagram_socket && other);
```

This assignment operator moves a datagram socket from one object to another.

Parameters

other The other **basic_datagram_socket** object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_datagram_socket(io_service&)` constructor.

5.68.47.2 `basic_datagram_socket::operator=` (2 of 2 overloads)

Move-assign a `basic_datagram_socket` from a socket of another protocol type.

```
template<
    typename Protocol,
    typename DatagramSocketService1>
enable_if< is_convertible< Protocol, Protocol >::value, basic_datagram_socket >::type & ←
    operator=(  
    basic_datagram_socket< Protocol, DatagramSocketService1 > && other);
```

This assignment operator moves a datagram socket from one object to another.

Parameters

other The other `basic_datagram_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_datagram_socket(io_service&)` constructor.

5.68.48 `basic_datagram_socket::protocol_type`

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.49 `basic_datagram_socket::receive`

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers);  
  
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,  
    socket_base::message_flags flags);  
  
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,  
    socket_base::message_flags flags,  
    asio::error_code & ec);
```

5.68.49.1 basic_datagram_socket::receive (1 of 3 overloads)

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers);
```

This function is used to receive data on the datagram socket. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

Return Value

The number of bytes received.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The receive operation can only be used with a connected socket. Use the `receive_from` function to receive data on an unconnected datagram socket.

Example

To receive into a single data buffer use the `buffer` function as follows:

```
socket.receive(asio::buffer(data, size));
```

See the `buffer` documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.68.49.2 basic_datagram_socket::receive (2 of 3 overloads)

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags);
```

This function is used to receive data on the datagram socket. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

flags Flags specifying how the receive call is to be made.

Return Value

The number of bytes received.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The receive operation can only be used with a connected socket. Use the `receive_from` function to receive data on an unconnected datagram socket.

5.68.49.3 `basic_datagram_socket::receive` (3 of 3 overloads)

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to receive data on the datagram socket. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

flags Flags specifying how the receive call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes received.

Remarks

The receive operation can only be used with a connected socket. Use the `receive_from` function to receive data on an unconnected datagram socket.

5.68.50 `basic_datagram_socket::receive_buffer_size`

Inherited from `socket_base`.

Socket option for the receive buffer size of a socket.

```
typedef implementation_defined receive_buffer_size;
```

Implements the SOL_SOCKET/SO_RCVBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.51 basic_datagram_socket::receive_from

Receive a datagram with the endpoint of the sender.

```
template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint);

template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags);

template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.68.51.1 basic_datagram_socket::receive_from (1 of 3 overloads)

Receive a datagram with the endpoint of the sender.

```
template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint);
```

This function is used to receive a datagram. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

sender_endpoint An endpoint object that receives the endpoint of the remote sender of the datagram.

Return Value

The number of bytes received.

Exceptions

asio::system_error Thrown on failure.

Example

To receive into a single data buffer use the **buffer** function as follows:

```
asio::ip::udp::endpoint sender_endpoint;
socket.receive_from(
    asio::buffer(data, size), sender_endpoint);
```

See the **buffer** documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

5.68.51.2 basic_datagram_socket::receive_from (2 of 3 overloads)

Receive a datagram with the endpoint of the sender.

```
template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags);
```

This function is used to receive a datagram. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

sender_endpoint An endpoint object that receives the endpoint of the remote sender of the datagram.

flags Flags specifying how the receive call is to be made.

Return Value

The number of bytes received.

Exceptions

asio::system_error Thrown on failure.

5.68.51.3 basic_datagram_socket::receive_from (3 of 3 overloads)

Receive a datagram with the endpoint of the sender.

```
template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to receive a datagram. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

sender_endpoint An endpoint object that receives the endpoint of the remote sender of the datagram.

flags Flags specifying how the receive call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes received.

5.68.52 basic_datagram_socket::receive_low_watermark

Inherited from socket_base.

Socket option for the receive low watermark.

```
typedef implementation_defined receive_low_watermark;
```

Implements the SOL_SOCKET/SO_RCVLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.53 basic_datagram_socket::remote_endpoint

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;  
  
endpoint_type remote_endpoint(  
    asio::error_code & ec) const;
```

5.68.53.1 basic_datagram_socket::remote_endpoint (1 of 2 overloads)

Inherited from basic_socket.

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;
```

This function is used to obtain the remote endpoint of the socket.

Return Value

An object that represents the remote endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);  
...  
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint();
```

5.68.53.2 basic_datagram_socket::remote_endpoint (2 of 2 overloads)

Inherited from basic_socket.

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint(  
    asio::error_code & ec) const;
```

This function is used to obtain the remote endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the remote endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.68.54 basic_datagram_socket::reuse_address

Inherited from socket_base.

Socket option to allow the socket to be bound to an address that is already in use.

```
typedef implementation_defined reuse_address;
```

Implements the SOL_SOCKET/SO_REUSEADDR socket option.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.55 basic_datagram_socket::send

Send some data on a connected socket.

```

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags);

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);

```

5.68.55.1 basic_datagram_socket::send (1 of 3 overloads)

Send some data on a connected socket.

```

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers);

```

This function is used to send data on the datagram socket. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One ore more data buffers to be sent on the socket.

Return Value

The number of bytes sent.

Exceptions

asio::system_error Thrown on failure.

Remarks

The send operation can only be used with a connected socket. Use the `send_to` function to send data on an unconnected datagram socket.

Example

To send a single data buffer use the `buffer` function as follows:

```
socket.send(asio::buffer(data, size));
```

See the `buffer` documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.68.55.2 basic_datagram_socket::send (2 of 3 overloads)

Send some data on a connected socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags);
```

This function is used to send data on the datagram socket. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One ore more data buffers to be sent on the socket.

flags Flags specifying how the send call is to be made.

Return Value

The number of bytes sent.

Exceptions

asio::system_error Thrown on failure.

Remarks

The send operation can only be used with a connected socket. Use the `send_to` function to send data on an unconnected datagram socket.

5.68.55.3 basic_datagram_socket::send (3 of 3 overloads)

Send some data on a connected socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to send data on the datagram socket. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One or more data buffers to be sent on the socket.

flags Flags specifying how the send call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes sent.

Remarks

The send operation can only be used with a connected socket. Use the `send_to` function to send data on an unconnected datagram socket.

5.68.56 `basic_datagram_socket::send_buffer_size`

Inherited from socket_base.

Socket option for the send buffer size of a socket.

```
typedef implementation_defined send_buffer_size;
```

Implements the SOL_SOCKET/SO_SNDBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: `asio/basic_datagram_socket.hpp`

Convenience header: `asio.hpp`

5.68.57 `basic_datagram_socket::send_low_watermark`

Inherited from socket_base.

Socket option for the send low watermark.

```
typedef implementation_defined send_low_watermark;
```

Implements the SOL_SOCKET/SO SNDLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.58 basic_datagram_socket::send_to

Send a datagram to the specified endpoint.

```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination);

template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags);

template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.68.58.1 basic_datagram_socket::send_to (1 of 3 overloads)

Send a datagram to the specified endpoint.

```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination);
```

This function is used to send a datagram to the specified remote endpoint. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One or more data buffers to be sent to the remote endpoint.

destination The remote endpoint to which the data will be sent.

Return Value

The number of bytes sent.

Exceptions

asio::system_error Thrown on failure.

Example

To send a single data buffer use the [buffer](#) function as follows:

```
asio::ip::udp::endpoint destination(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.send_to(asio::buffer(data, size), destination);
```

See the [buffer](#) documentation for information on sending multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

5.68.58.2 basic_datagram_socket::send_to (2 of 3 overloads)

Send a datagram to the specified endpoint.

```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags);
```

This function is used to send a datagram to the specified remote endpoint. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One or more data buffers to be sent to the remote endpoint.

destination The remote endpoint to which the data will be sent.

flags Flags specifying how the send call is to be made.

Return Value

The number of bytes sent.

Exceptions

asio::system_error Thrown on failure.

5.68.58.3 **basic_datagram_socket::send_to** (3 of 3 overloads)

Send a datagram to the specified endpoint.

```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to send a datagram to the specified remote endpoint. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One or more data buffers to be sent to the remote endpoint.

destination The remote endpoint to which the data will be sent.

flags Flags specifying how the send call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes sent.

5.68.59 **basic_datagram_socket::service**

Inherited from basic_io_object.

(Deprecated: Use `get_service()`.) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.68.60 **basic_datagram_socket::service_type**

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef DatagramSocketService service_type;
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.61 basic_datagram_socket::set_option

Set an option on the socket.

```
void set_option(
    const SettableSocketOption & option);

asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.68.61.1 basic_datagram_socket::set_option (1 of 2 overloads)

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.68.61.2 basic_datagram_socket::set_option (2 of 2 overloads)

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

ec Set to indicate what error occurred, if any.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.68.62 basic_datagram_socket::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.68.62.1 basic_datagram_socket::shutdown (1 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

Exceptions

asio::system_error Thrown on failure.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
socket.shutdown(asio::ip::tcp::socket::shutdown_send);
```

5.68.62.2 basic_datagram_socket::shutdown (2 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

ec Set to indicate what error occurred, if any.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);
if (ec)
{
    // An error occurred.
}
```

5.68.63 basic_datagram_socket::shutdown_type

Inherited from socket_base.

Different ways a socket may be shutdown.

```
enum shutdown_type
```

Values

shutdown_receive Shutdown the receive side of the socket.

shutdown_send Shutdown the send side of the socket.

shutdown_both Shutdown both send and receive on the socket.

5.69 basic_deadline_timer

Provides waitable timer functionality.

```
template<
    typename Time,
    typename TimeTraits = asio::time_traits<Time>,
    typename TimerService = deadline_timer_service<Time, TimeTraits>>
class basic_deadline_timer :
    public basic_io_object< TimerService >
```

Types

Name	Description
duration_type	The duration type.
implementation_type	The underlying implementation type of I/O object.
service_type	The type of the service that will be used to provide I/O operations.
time_type	The time type.
traits_type	The time traits type.

Member Functions

Name	Description
async_wait	Start an asynchronous wait on the timer.
basic_deadline_timer	Constructor. Constructor to set a particular expiry time as an absolute time. Constructor to set a particular expiry time relative to now.
cancel	Cancel any asynchronous operations that are waiting on the timer.
cancel_one	Cancels one asynchronous operation that is waiting on the timer.
expires_at	Get the timer's expiry time as an absolute time. Set the timer's expiry time as an absolute time.

Name	Description
expires_from_now	Get the timer's expiry time relative to now. Set the timer's expiry time relative to now.
get_io_service	Get the io_service associated with the object.
wait	Perform a blocking wait on the timer.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_deadline_timer` class template provides the ability to perform a blocking or asynchronous wait for a timer to expire.

A deadline timer is always in one of two states: "expired" or "not expired". If the `wait()` or `async_wait()` function is called on an expired timer, the wait operation will complete immediately.

Most applications will use the `deadline_timer` typedef.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Examples

Performing a blocking wait:

```
// Construct a timer without setting an expiry time.
asio::deadline_timer timer(io_service);

// Set an expiry time relative to now.
timer.expires_from_now(boost::posix_time::seconds(5));

// Wait for the timer to expire.
timer.wait();
```

Performing an asynchronous wait:

```
void handler(const asio::error_code& error)
{
    if (!error)
    {
        // Timer expired.
    }
}

...

// Construct a timer with an absolute expiry time.
asio::deadline_timer timer(io_service,
    boost::posix_time::time_from_string("2005-12-07 23:59:59.000"));

// Start an asynchronous wait.
timer.async_wait(handler);
```

Changing an active deadline_timer's expiry time

Changing the expiry time of a timer while there are pending asynchronous waits causes those wait operations to be cancelled. To ensure that the action associated with the timer is performed only once, use something like this: used:

```
void on_some_event()
{
    if (my_timer.expires_from_now(seconds(5)) > 0)
    {
        // We managed to cancel the timer. Start new asynchronous wait.
        my_timer.async_wait(on_timeout);
    }
    else
    {
        // Too late, timer has already expired!
    }
}

void on_timeout(const asio::error_code& e)
{
    if (e != asio::error::operation_aborted)
    {
        // Timer was not cancelled, take necessary action.
    }
}
```

- The `asio::basic_deadline_timer::expires_from_now()` function cancels any pending asynchronous waits, and returns the number of asynchronous waits that were cancelled. If it returns 0 then you were too late and the wait handler has already been executed, or will soon be executed. If it returns 1 then the wait handler was successfully cancelled.
- If a wait handler is cancelled, the `error_code` passed to it contains the value `asio::error::operation_aborted`.

Requirements

Header: `asio/basic_deadline_timer.hpp`

Convenience header: `asio.hpp`

5.69.1 basic_deadline_timer::async_wait

Start an asynchronous wait on the timer.

```
template<
    typename WaitHandler>
void-or-deduced async_wait(
    WaitHandler handler);
```

This function may be used to initiate an asynchronous wait against the timer. It always returns immediately.

For each call to `async_wait()`, the supplied handler will be called exactly once. The handler will be called when:

- The timer has expired.
- The timer was cancelled, in which case the handler is passed the error code `asio::error::operation_aborted`.

Parameters

handler The handler to be called when the timer expires. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error // Result of operation.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.69.2 basic_deadline_timer::basic_deadline_timer

Constructor.

```
explicit basic_deadline_timer(
    asio::io_service & io_service);
```

Constructor to set a particular expiry time as an absolute time.

```
basic_deadline_timer(
    asio::io_service & io_service,
    const time_type & expiry_time);
```

Constructor to set a particular expiry time relative to now.

```
basic_deadline_timer(
    asio::io_service & io_service,
    const duration_type & expiry_time);
```

5.69.2.1 basic_deadline_timer::basic_deadline_timer (1 of 3 overloads)

Constructor.

```
basic_deadline_timer(
    asio::io_service & io_service);
```

This constructor creates a timer without setting an expiry time. The `expires_at()` or `expires_from_now()` functions must be called to set an expiry time before the timer can be waited on.

Parameters

io_service The `io_service` object that the timer will use to dispatch handlers for any asynchronous operations performed on the timer.

5.69.2.2 `basic_deadline_timer::basic_deadline_timer (2 of 3 overloads)`

Constructor to set a particular expiry time as an absolute time.

```
basic_deadline_timer(
    asio::io_service & io_service,
    const time_type & expiry_time);
```

This constructor creates a timer and sets the expiry time.

Parameters

io_service The `io_service` object that the timer will use to dispatch handlers for any asynchronous operations performed on the timer.

expiry_time The expiry time to be used for the timer, expressed as an absolute time.

5.69.2.3 `basic_deadline_timer::basic_deadline_timer (3 of 3 overloads)`

Constructor to set a particular expiry time relative to now.

```
basic_deadline_timer(
    asio::io_service & io_service,
    const duration_type & expiry_time);
```

This constructor creates a timer and sets the expiry time.

Parameters

io_service The `io_service` object that the timer will use to dispatch handlers for any asynchronous operations performed on the timer.

expiry_time The expiry time to be used for the timer, relative to now.

5.69.3 `basic_deadline_timer::cancel`

Cancel any asynchronous operations that are waiting on the timer.

```
std::size_t cancel();  
  
std::size_t cancel(  
    asio::error_code & ec);
```

5.69.3.1 `basic_deadline_timer::cancel (1 of 2 overloads)`

Cancel any asynchronous operations that are waiting on the timer.

```
std::size_t cancel();
```

This function forces the completion of any pending asynchronous wait operations against the timer. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Cancelling the timer does not change the expiry time.

Return Value

The number of asynchronous operations that were cancelled.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

If the timer has already expired when `cancel()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.69.3.2 `basic_deadline_timer::cancel (2 of 2 overloads)`

Cancel any asynchronous operations that are waiting on the timer.

```
std::size_t cancel(  
    asio::error_code & ec);
```

This function forces the completion of any pending asynchronous wait operations against the timer. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Cancelling the timer does not change the expiry time.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of asynchronous operations that were cancelled.

Remarks

If the timer has already expired when `cancel()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.69.4 basic_deadline_timer::cancel_one

Cancels one asynchronous operation that is waiting on the timer.

```
std::size_t cancel_one();  
  
std::size_t cancel_one(  
    asio::error_code & ec);
```

5.69.4.1 basic_deadline_timer::cancel_one (1 of 2 overloads)

Cancels one asynchronous operation that is waiting on the timer.

```
std::size_t cancel_one();
```

This function forces the completion of one pending asynchronous wait operation against the timer. Handlers are cancelled in FIFO order. The handler for the cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Cancelling the timer does not change the expiry time.

Return Value

The number of asynchronous operations that were cancelled. That is, either 0 or 1.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

If the timer has already expired when `cancel_one()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.69.4.2 basic_deadline_timer::cancel_one (2 of 2 overloads)

Cancels one asynchronous operation that is waiting on the timer.

```
std::size_t cancel_one(  
    asio::error_code & ec);
```

This function forces the completion of one pending asynchronous wait operation against the timer. Handlers are cancelled in FIFO order. The handler for the cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Cancelling the timer does not change the expiry time.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of asynchronous operations that were cancelled. That is, either 0 or 1.

Remarks

If the timer has already expired when `cancel_one()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.69.5 `basic_deadline_timer::duration_type`

The duration type.

```
typedef traits_type::duration_type duration_type;
```

Requirements

Header: `asio/basic_deadline_timer.hpp`

Convenience header: `asio.hpp`

5.69.6 `basic_deadline_timer::expires_at`

Get the timer's expiry time as an absolute time.

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

Set the timer's expiry time as an absolute time.

```
std::size_t expires_at(
    const time_type & expiry_time);

std::size_t expires_at(
    const time_type & expiry_time,
    asio::error_code & ec);
```

5.69.6.1 `basic_deadline_timer::expires_at (1 of 3 overloads)`

Get the timer's expiry time as an absolute time.

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

This function may be used to obtain the timer's current expiry time. Whether the timer has expired or not does not affect this value.

5.69.6.2 basic_deadline_timer::expires_at (2 of 3 overloads)

Set the timer's expiry time as an absolute time.

```
std::size_t expires_at(
    const time_type & expiry_time);
```

This function sets the expiry time. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Parameters

expiry_time The expiry time to be used for the timer.

Return Value

The number of asynchronous operations that were cancelled.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

If the timer has already expired when `expires_at()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.69.6.3 basic_deadline_timer::expires_at (3 of 3 overloads)

Set the timer's expiry time as an absolute time.

```
std::size_t expires_at(
    const time_type & expiry_time,
    asio::error_code & ec);
```

This function sets the expiry time. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Parameters

expiry_time The expiry time to be used for the timer.

ec Set to indicate what error occurred, if any.

Return Value

The number of asynchronous operations that were cancelled.

Remarks

If the timer has already expired when `expires_at()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.69.7 `basic_deadline_timer::expires_from_now`

Get the timer's expiry time relative to now.

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

Set the timer's expiry time relative to now.

```
std::size_t expires_from_now(
    const duration_type & expiry_time);

std::size_t expires_from_now(
    const duration_type & expiry_time,
    asio::error_code & ec);
```

5.69.7.1 `basic_deadline_timer::expires_from_now (1 of 3 overloads)`

Get the timer's expiry time relative to now.

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

This function may be used to obtain the timer's current expiry time. Whether the timer has expired or not does not affect this value.

5.69.7.2 `basic_deadline_timer::expires_from_now (2 of 3 overloads)`

Set the timer's expiry time relative to now.

```
std::size_t expires_from_now(
    const duration_type & expiry_time);
```

This function sets the expiry time. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Parameters

expiry_time The expiry time to be used for the timer.

Return Value

The number of asynchronous operations that were cancelled.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

If the timer has already expired when `expires_from_now()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.69.7.3 `basic_deadline_timer::expires_from_now (3 of 3 overloads)`

Set the timer's expiry time relative to now.

```
std::size_t expires_from_now(
    const duration_type & expiry_time,
    asio::error_code & ec);
```

This function sets the expiry time. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Parameters

expiry_time The expiry time to be used for the timer.

ec Set to indicate what error occurred, if any.

Return Value

The number of asynchronous operations that were cancelled.

Remarks

If the timer has already expired when `expires_from_now()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.69.8 `basic_deadline_timer::get_implementation`

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();

const implementation_type & get_implementation() const;
```

5.69.8.1 basic_deadline_timer::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.69.8.2 basic_deadline_timer::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.69.9 basic_deadline_timer::get_io_service

Inherited from basic_io_object.

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the **io_service** object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the **io_service** object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.69.10 basic_deadline_timer::get_service

Get the service associated with the I/O object.

```
service_type & get_service();
```

```
const service_type & get_service() const;
```

5.69.10.1 basic_deadline_timer::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.69.10.2 basic_deadline_timer::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.69.11 basic_deadline_timer::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.69.12 basic_deadline_timer::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_deadline_timer.hpp

Convenience header: asio.hpp

5.69.13 basic_deadline_timer::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`.) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.69.14 basic_deadline_timer::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef TimerService service_type;
```

Requirements

Header: asio/basic_deadline_timer.hpp

Convenience header: asio.hpp

5.69.15 basic_deadline_timer::time_type

The time type.

```
typedef traits_type::time_type time_type;
```

Requirements

Header: asio/basic_deadline_timer.hpp

Convenience header: asio.hpp

5.69.16 basic_deadline_timer::traits_type

The time traits type.

```
typedef TimeTraits traits_type;
```

Requirements

Header: asio/basic_deadline_timer.hpp

Convenience header: asio.hpp

5.69.17 basic_deadline_timer::wait

Perform a blocking wait on the timer.

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

5.69.17.1 basic_deadline_timer::wait (1 of 2 overloads)

Perform a blocking wait on the timer.

```
void wait();
```

This function is used to wait for the timer to expire. This function blocks and does not return until the timer has expired.

Exceptions

asio::system_error Thrown on failure.

5.69.17.2 basic_deadline_timer::wait (2 of 2 overloads)

Perform a blocking wait on the timer.

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

This function is used to wait for the timer to expire. This function blocks and does not return until the timer has expired.

Parameters

ec Set to indicate what error occurred, if any.

5.70 basic_io_object

Base class for all I/O objects.

```
template<
    typename IoObjectService>
class basic_io_object
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
get_io_service	Get the io_service associated with the object.

Protected Member Functions

Name	Description
basic_io_object	Construct a basic_io_object. Move-construct a basic_io_object.
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
operator=	Move-assign a basic_io_object.
~basic_io_object	Protected destructor to prevent deletion through this type.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

Remarks

All I/O objects are non-copyable. However, when using C++0x, certain I/O objects do support move construction and move assignment.

Requirements

Header: asio/basic_io_object.hpp

Convenience header: asio.hpp

5.70.1 basic_io_object::basic_io_object

Construct a [basic_io_object](#).

```
explicit basic_io_object(  
    asio::io_service & io_service);
```

Move-construct a [basic_io_object](#).

```
basic_io_object(  
    basic_io_object && other);
```

5.70.1.1 basic_io_object::basic_io_object (1 of 2 overloads)

Construct a [basic_io_object](#).

```
basic_io_object(  
    asio::io_service & io_service);
```

Performs:

```
get_service().construct(get_implementation());
```

5.70.1.2 basic_io_object::basic_io_object (2 of 2 overloads)

Move-construct a [basic_io_object](#).

```
basic_io_object(  
    basic_io_object && other);
```

Performs:

```
get_service().move_construct(  
    get_implementation(), other.get_implementation());
```

Remarks

Available only for services that support movability,

5.70.2 basic_io_object::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.70.2.1 basic_io_object::get_implementation (1 of 2 overloads)

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.70.2.2 basic_io_object::get_implementation (2 of 2 overloads)

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.70.3 basic_io_object::get_io_service

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the **io_service** object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the **io_service** object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.70.4 basic_io_object::get_service

Get the service associated with the I/O object.

```
service_type & get_service();  
  
const service_type & get_service() const;
```

5.70.4.1 basic_io_object::get_service (1 of 2 overloads)

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.70.4.2 basic_io_object::get_service (2 of 2 overloads)

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.70.5 basic_io_object::implementation

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.70.6 basic_io_object::implementation_type

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: `asio/basic_io_object.hpp`

Convenience header: `asio.hpp`

5.70.7 basic_io_object::operator=

Move-assign a `basic_io_object`.

```
basic_io_object & operator=(  
    basic_io_object && other);
```

Performs:

```
get_service().move_assign(get_implementation(),  
    other.get_service(), other.get_implementation());
```

Remarks

Available only for services that support movability,

5.70.8 basic_io_object::service

(Deprecated: Use `get_service()`.) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.70.9 basic_io_object::service_type

The type of the service that will be used to provide I/O operations.

```
typedef IoObjectService service_type;
```

Requirements

Header: asio/basic_io_object.hpp

Convenience header: asio.hpp

5.70.10 basic_io_object::~basic_io_object

Protected destructor to prevent deletion through this type.

```
~basic_io_object();
```

Performs:

```
get_service().destroy(get_implementation());
```

5.71 basic_raw_socket

Provides raw-oriented socket functionality.

```
template<
    typename Protocol,
    typename RawSocketService = raw_socket_service<Protocol>>
class basic_raw_socket :
    public basic_socket< Protocol, RawSocketService >
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.

Name	Description
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive on a connected socket.
async_receive_from	Start an asynchronous receive.
async_send	Start an asynchronous send on a connected socket.
async_send_to	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.

Name	Description
<code>basic_raw_socket</code>	Construct a basic_raw_socket without opening it. Construct and open a basic_raw_socket. Construct a basic_raw_socket, opening it and binding it to the given local endpoint. Construct a basic_raw_socket on an existing native socket. Move-construct a basic_raw_socket from another. Move-construct a basic_raw_socket from a socket of another protocol type.
<code>bind</code>	Bind the socket to the given local endpoint.
<code>cancel</code>	Cancel all asynchronous operations associated with the socket.
<code>close</code>	Close the socket.
<code>connect</code>	Connect the socket to the specified endpoint.
<code>get_io_service</code>	Get the io_service associated with the object.
<code>get_option</code>	Get an option from the socket.
<code>io_control</code>	Perform an IO control command on the socket.
<code>is_open</code>	Determine whether the socket is open.
<code>local_endpoint</code>	Get the local endpoint of the socket.
<code>lowest_layer</code>	Get a reference to the lowest layer. Get a const reference to the lowest layer.
<code>native</code>	(Deprecated: Use native_handle().) Get the native socket representation.
<code>native_handle</code>	Get the native socket representation.
<code>native_non_blocking</code>	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
<code>non_blocking</code>	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
<code>open</code>	Open the socket using the specified protocol.
<code>operator=</code>	Move-assign a basic_raw_socket from another. Move-assign a basic_raw_socket from a socket of another protocol type.
<code>receive</code>	Receive some data on a connected socket.
<code>receive_from</code>	Receive raw data with the endpoint of the sender.

Name	Description
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on a connected socket.
send_to	Send raw data to the specified endpoint.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_raw_socket` class template provides asynchronous and blocking raw-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.1 basic_raw_socket::assign

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);

asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.71.1.1 basic_raw_socket::assign (1 of 2 overloads)

Inherited from basic_socket.

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

5.71.1.2 basic_raw_socket::assign (2 of 2 overloads)

Inherited from basic_socket.

Assign an existing native socket to the socket.

```
asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.71.2 basic_raw_socket::async_connect

Inherited from basic_socket.

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
void-or-deduced async_connect(
    const endpoint_type & peer_endpoint,
    ConnectHandler handler);
```

This function is used to asynchronously connect a socket to the specified remote endpoint. The function call always returns immediately.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected. Copies will be made of the endpoint object as required.

handler The handler to be called when the connection operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(  
    const asio::error_code& error // Result of operation  
) ;
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

```
void connect_handler(const asio::error_code& error)  
{  
    if (!error)  
    {  
        // Connect succeeded.  
    }  
}  
  
...  
  
asio::ip::tcp::socket socket(io_service);  
asio::ip::tcp::endpoint endpoint(  
    asio::ip::address::from_string("1.2.3.4"), 12345);  
socket.async_connect(endpoint, connect_handler);
```

5.71.3 basic_raw_socket::async_receive

Start an asynchronous receive on a connected socket.

```
template<  
    typename MutableBufferSequence,  
    typename ReadHandler>  
void-or-deduced async_receive(  
    const MutableBufferSequence & buffers,  
    ReadHandler handler);  
  
template<  
    typename MutableBufferSequence,  
    typename ReadHandler>  
void-or-deduced async_receive(  
    const MutableBufferSequence & buffers,  
    socket_base::message_flags flags,  
    ReadHandler handler);
```

5.71.3.1 basic_raw_socket::async_receive (1 of 2 overloads)

Start an asynchronous receive on a connected socket.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

This function is used to asynchronously receive data from the raw socket. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The `async_receive` operation can only be used with a connected socket. Use the `async_receive_from` function to receive data on an unconnected raw socket.

Example

To receive into a single data buffer use the `buffer` function as follows:

```
socket.async_receive(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.71.3.2 basic_raw_socket::async_receive (2 of 2 overloads)

Start an asynchronous receive on a connected socket.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    ReadHandler handler);
```

This function is used to asynchronously receive data from the raw socket. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

flags Flags specifying how the receive call is to be made.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The `async_receive` operation can only be used with a connected socket. Use the `async_receive_from` function to receive data on an unconnected raw socket.

5.71.4 basic_raw_socket::async_receive_from

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    ReadHandler handler);

template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    ReadHandler handler);
```

5.71.4.1 basic_raw_socket::async_receive_from (1 of 2 overloads)

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    ReadHandler handler);
```

This function is used to asynchronously receive raw data. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

sender_endpoint An endpoint object that receives the endpoint of the remote sender of the data. Ownership of the sender_endpoint object is retained by the caller, which must guarantee that it is valid until the handler is called.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To receive into a single data buffer use the **buffer** function as follows:

```
socket.async_receive_from(
    asio::buffer(data, size), 0, sender_endpoint, handler);
```

See the **buffer** documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.71.4.2 basic_raw_socket::async_receive_from (2 of 2 overloads)

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    ReadHandler handler);
```

This function is used to asynchronously receive raw data. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

sender_endpoint An endpoint object that receives the endpoint of the remote sender of the data. Ownership of the sender_endpoint object is retained by the caller, which must guarantee that it is valid until the handler is called.

flags Flags specifying how the receive call is to be made.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.71.5 basic_raw_socket::async_send

Start an asynchronous send on a connected socket.

```

template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    WriteHandler handler);

template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);

```

5.71.5.1 basic_raw_socket::async_send (1 of 2 overloads)

Start an asynchronous send on a connected socket.

```

template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    WriteHandler handler);

```

This function is used to send data on the raw socket. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One or more data buffers to be sent on the socket. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes sent.
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The `async_send` operation can only be used with a connected socket. Use the `async_send_to` function to send data on an unconnected raw socket.

Example

To send a single data buffer use the `buffer` function as follows:

```
socket.async_send(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.71.5.2 basic_raw_socket::async_send (2 of 2 overloads)

Start an asynchronous send on a connected socket.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);
```

This function is used to send data on the raw socket. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One or more data buffers to be sent on the socket. Although the `buffers` object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

flags Flags specifying how the send call is to be made.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes sent.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The `async_send` operation can only be used with a connected socket. Use the `async_send_to` function to send data on an unconnected raw socket.

5.71.6 basic_raw_socket::async_send_to

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    WriteHandler handler);

template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    WriteHandler handler);
```

5.71.6.1 basic_raw_socket::async_send_to (1 of 2 overloads)

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    WriteHandler handler);
```

This function is used to asynchronously send raw data to the specified remote endpoint. The function call always returns immediately.

Parameters

buffers One or more data buffers to be sent to the remote endpoint. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

destination The remote endpoint to which the data will be sent. Copies will be made of the endpoint as required.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes sent.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To send a single data buffer use the **buffer** function as follows:

```
asio::ip::udp::endpoint destination(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.async_send_to(
    asio::buffer(data, size), destination, handler);
```

See the **buffer** documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.71.6.2 basic_raw_socket::async_send_to (2 of 2 overloads)

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    WriteHandler handler);
```

This function is used to asynchronously send raw data to the specified remote endpoint. The function call always returns immediately.

Parameters

buffers One or more data buffers to be sent to the remote endpoint. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

flags Flags specifying how the send call is to be made.

destination The remote endpoint to which the data will be sent. Copies will be made of the endpoint as required.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes sent.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.71.7 basic_raw_socket::at_mark

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;

bool at_mark(
    asio::error_code & ec) const;
```

5.71.7.1 basic_raw_socket::at_mark (1 of 2 overloads)

Inherited from basic_socket.

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

Exceptions

`asio::system_error` Thrown on failure.

5.71.7.2 basic_raw_socket::at_mark (2 of 2 overloads)

Inherited from basic_socket.

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark(
    asio::error_code & ec) const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Parameters

`ec` Set to indicate what error occurred, if any.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

5.71.8 basic_raw_socket::available

Determine the number of bytes available for reading.

```
std::size_t available() const;
```

```
std::size_t available(
    asio::error_code & ec) const;
```

5.71.8.1 basic_raw_socket::available (1 of 2 overloads)

Inherited from basic_socket.

Determine the number of bytes available for reading.

```
std::size_t available() const;
```

This function is used to determine the number of bytes that may be read without blocking.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

Exceptions

`asio::system_error` Thrown on failure.

5.71.8.2 `basic_raw_socket::available` (2 of 2 overloads)

Inherited from basic_socket.

Determine the number of bytes available for reading.

```
std::size_t available(
    asio::error_code & ec) const;
```

This function is used to determine the number of bytes that may be read without blocking.

Parameters

`ec` Set to indicate what error occurred, if any.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

5.71.9 `basic_raw_socket::basic_raw_socket`

Construct a `basic_raw_socket` without opening it.

```
explicit basic_raw_socket(
    asio::io_service & io_service);
```

Construct and open a `basic_raw_socket`.

```
basic_raw_socket(
    asio::io_service & io_service,
    const protocol_type & protocol);
```

Construct a `basic_raw_socket`, opening it and binding it to the given local endpoint.

```
basic_raw_socket(
    asio::io_service & io_service,
    const endpoint_type & endpoint);
```

Construct a `basic_raw_socket` on an existing native socket.

```
basic_raw_socket(
    asio::io_service & io_service,
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

Move-construct a `basic_raw_socket` from another.

```
basic_raw_socket(
    basic_raw_socket && other);
```

Move-construct a `basic_raw_socket` from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename RawSocketService1>
basic_raw_socket(
    basic_raw_socket< Protocol1, RawSocketService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.71.9.1 `basic_raw_socket::basic_raw_socket (1 of 6 overloads)`

Construct a `basic_raw_socket` without opening it.

```
basic_raw_socket(
   asio::io_service & io_service);
```

This constructor creates a raw socket without opening it. The `open()` function must be called before data can be sent or received on the socket.

Parameters

io_service The `io_service` object that the raw socket will use to dispatch handlers for any asynchronous operations performed on the socket.

5.71.9.2 `basic_raw_socket::basic_raw_socket (2 of 6 overloads)`

Construct and open a `basic_raw_socket`.

```
basic_raw_socket(
    asio::io_service & io_service,
    const protocol_type & protocol);
```

This constructor creates and opens a raw socket.

Parameters

io_service The `io_service` object that the raw socket will use to dispatch handlers for any asynchronous operations performed on the socket.

protocol An object specifying protocol parameters to be used.

Exceptions

`asio::system_error` Thrown on failure.

5.71.9.3 basic_raw_socket::basic_raw_socket (3 of 6 overloads)

Construct a `basic_raw_socket`, opening it and binding it to the given local endpoint.

```
basic_raw_socket(
    asio::io_service & io_service,
    const endpoint_type & endpoint);
```

This constructor creates a raw socket and automatically opens it bound to the specified endpoint on the local machine. The protocol used is the protocol associated with the given endpoint.

Parameters

io_service The `io_service` object that the raw socket will use to dispatch handlers for any asynchronous operations performed on the socket.

endpoint An endpoint on the local machine to which the raw socket will be bound.

Exceptions

`asio::system_error` Thrown on failure.

5.71.9.4 basic_raw_socket::basic_raw_socket (4 of 6 overloads)

Construct a `basic_raw_socket` on an existing native socket.

```
basic_raw_socket(
    asio::io_service & io_service,
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

This constructor creates a raw socket object to hold an existing native socket.

Parameters

io_service The `io_service` object that the raw socket will use to dispatch handlers for any asynchronous operations performed on the socket.

protocol An object specifying protocol parameters to be used.

native_socket The new underlying socket implementation.

Exceptions

`asio::system_error` Thrown on failure.

5.71.9.5 basic_raw_socket::basic_raw_socket (5 of 6 overloads)

Move-construct a `basic_raw_socket` from another.

```
basic_raw_socket(
    basic_raw_socket && other);
```

This constructor moves a raw socket from one object to another.

Parameters

other The other `basic_raw_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_raw_socket(io_service&)` constructor.

5.71.9.6 `basic_raw_socket::basic_raw_socket (6 of 6 overloads)`

Move-construct a `basic_raw_socket` from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename RawSocketService1>
basic_raw_socket(
    basic_raw_socket< Protocol1, RawSocketService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

This constructor moves a raw socket from one object to another.

Parameters

other The other `basic_raw_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_raw_socket(io_service&)` constructor.

5.71.10 `basic_raw_socket::bind`

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.71.10.1 `basic_raw_socket::bind (1 of 2 overloads)`

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345));
```

5.71.10.2 basic_raw_socket::bind (2 of 2 overloads)

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
asio::error_code ec;
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345), ec);
if (ec)
{
    // An error occurred.
}
```

5.71.11 basic_raw_socket::broadcast

Inherited from socket_base.

Socket option to permit sending of broadcast messages.

```
typedef implementation_defined broadcast;
```

Implements the SOL_SOCKET/SO_BROADCAST socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.12 basic_raw_socket::bytes_readable

Inherited from socket_base.

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::bytes_readable command(true);
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.13 basic_raw_socket::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.71.13.1 basic_raw_socket::cancel (1 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
void cancel();
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.71.13.2 basic_raw_socket::cancel (2 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.71.14 basic_raw_socket::close

Close the socket.

```
void close();  
  
asio::error_code close(  
    asio::error_code & ec);
```

5.71.14.1 basic_raw_socket::close (1 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
void close();
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

asio::system_error Thrown on failure. Note that, even if the function indicates an error, the underlying descriptor is closed.

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.71.14.2 basic_raw_socket::close (2 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
asio::error_code close(  
    asio::error_code & ec);
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any. Note that, even if the function indicates an error, the underlying descriptor is closed.

Example

```
asio::ip::tcp::socket socket(io_service);  
...  
asio::error_code ec;  
socket.close(ec);  
if (ec)  
{  
    // An error occurred.  
}
```

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.71.15 basic_raw_socket::connect

Connect the socket to the specified endpoint.

```
void connect(  
    const endpoint_type & peer_endpoint);  
  
asio::error_code connect(  
    const endpoint_type & peer_endpoint,  
    asio::error_code & ec);
```

5.71.15.1 basic_raw_socket::connect (1 of 2 overloads)

Inherited from basic_socket.

Connect the socket to the specified endpoint.

```
void connect(  
    const endpoint_type & peer_endpoint);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.connect(endpoint);
```

5.71.15.2 basic_raw_socket::connect (2 of 2 overloads)

Inherited from basic_socket.

Connect the socket to the specified endpoint.

```
asio::error_code connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
asio::error_code ec;
socket.connect(endpoint, ec);
if (ec)
{
    // An error occurred.
}
```

5.71.16 basic_raw_socket::debug

Inherited from socket_base.

Socket option to enable socket-level debugging.

```
typedef implementation_defined debug;
```

Implements the SOL_SOCKET/SO_DEBUG socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.17 basic_raw_socket::do_not_route

Inherited from socket_base.

Socket option to prevent routing, use local interfaces only.

```
typedef implementation_defined do_not_route;
```

Implements the SOL_SOCKET/SO_DONTROUTE socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.18 basic_raw_socket::enable_connection_aborted

Inherited from socket_base.

Socket option to report aborted connections on accept.

```
typedef implementation_defined enable_connection_aborted;
```

Implements a custom socket option that determines whether or not an accept operation is permitted to fail with `asio::error::connection_aborted`. By default the option is false.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.19 basic_raw_socket::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.20 basic_raw_socket::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();

const implementation_type & get_implementation() const;
```

5.71.20.1 basic_raw_socket::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.71.20.2 basic_raw_socket::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.71.21 basic_raw_socket::get_io_service

Inherited from basic_io_object.

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the **io_service** object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the **io_service** object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.71.22 basic_raw_socket::get_option

Get an option from the socket.

```
void get_option(
    GettableSocketOption & option) const;

asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.71.22.1 basic_raw_socket::get_option (1 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.71.22.2 basic_raw_socket::get_option (2 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
asio::error_code ec;
socket.get_option(option, ec);
if (ec)
{
    // An error occurred.
}
bool is_set = option.value();
```

5.71.23 basic_raw_socket::get_service

Get the service associated with the I/O object.

```
service_type & get_service();  
  
const service_type & get_service() const;
```

5.71.23.1 basic_raw_socket::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.71.23.2 basic_raw_socket::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.71.24 basic_raw_socket::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.71.25 basic_raw_socket::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.26 basic_raw_socket::io_control

Perform an IO control command on the socket.

```
void io_control(  
    IoControlCommand & command);
```

```
asio::error_code io_control(  
    IoControlCommand & command,  
    asio::error_code & ec);
```

5.71.26.1 basic_raw_socket::io_control (1 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

5.71.26.2 basic_raw_socket::io_control (2 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
asio::error_code ec;
socket.io_control(command, ec);
if (ec)
{
    // An error occurred.
}
std::size_t bytes_readable = command.get();
```

5.71.27 basic_raw_socket::is_open

Inherited from basic_socket.

Determine whether the socket is open.

```
bool is_open() const;
```

5.71.28 basic_raw_socket::keep_alive

Inherited from socket_base.

Socket option to send keep-alives.

```
typedef implementation_defined keep_alive;
```

Implements the SOL_SOCKET/SO_KEEPALIVE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.29 basic_raw_socket::linger

Inherited from socket_base.

Socket option to specify whether the socket lingers on close if unsent data is present.

```
typedef implementation_defined linger;
```

Implements the SOL_SOCKET/SO_LINGER socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option;
socket.get_option(option);
bool is_set = option.enabled();
unsigned short timeout = option.timeout();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.30 basic_raw_socket::local_endpoint

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;

endpoint_type local_endpoint(
    asio::error_code & ec) const;
```

5.71.30.1 basic_raw_socket::local_endpoint (1 of 2 overloads)

Inherited from basic_socket.

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;
```

This function is used to obtain the locally bound endpoint of the socket.

Return Value

An object that represents the local endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::endpoint endpoint = socket.local_endpoint();
```

5.71.30.2 basic_raw_socket::local_endpoint (2 of 2 overloads)

Inherited from basic_socket.

Get the local endpoint of the socket.

```
endpoint_type local_endpoint(
    asio::error_code & ec) const;
```

This function is used to obtain the locally bound endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the local endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.local_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.71.31 basic_raw_socket::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.71.31.1 basic_raw_socket::lowest_layer (1 of 2 overloads)

Inherited from basic_socket.

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a **basic_socket** cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.71.31.2 basic_raw_socket::lowest_layer (2 of 2 overloads)

Inherited from basic_socket.

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a **basic_socket** cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.71.32 basic_raw_socket::lowest_layer_type

Inherited from basic_socket.

A **basic_socket** is always the lowest layer.

```
typedef basic_socket< Protocol, RawSocketService > lowest_layer_type;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.

Name	Description
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.

Name	Description
<code>basic_socket</code>	Construct a <code>basic_socket</code> without opening it. Construct and open a <code>basic_socket</code> . Construct a <code>basic_socket</code> , opening it and binding it to the given local endpoint. Construct a <code>basic_socket</code> on an existing native socket. Move-construct a <code>basic_socket</code> from another. Move-construct a <code>basic_socket</code> from a socket of another protocol type.
<code>bind</code>	Bind the socket to the given local endpoint.
<code>cancel</code>	Cancel all asynchronous operations associated with the socket.
<code>close</code>	Close the socket.
<code>connect</code>	Connect the socket to the specified endpoint.
<code>get_io_service</code>	Get the <code>io_service</code> associated with the object.
<code>get_option</code>	Get an option from the socket.
<code>io_control</code>	Perform an IO control command on the socket.
<code>is_open</code>	Determine whether the socket is open.
<code>local_endpoint</code>	Get the local endpoint of the socket.
<code>lowest_layer</code>	Get a reference to the lowest layer. Get a const reference to the lowest layer.
<code>native</code>	(Deprecated: Use <code>native_handle()</code>) Get the native socket representation.
<code>native_handle</code>	Get the native socket representation.
<code>native_non_blocking</code>	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
<code>non_blocking</code>	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
<code>open</code>	Open the socket using the specified protocol.
<code>operator=</code>	Move-assign a <code>basic_socket</code> from another. Move-assign a <code>basic_socket</code> from a socket of another protocol type.
<code>remote_endpoint</code>	Get the remote endpoint of the socket.
<code>set_option</code>	Set an option on the socket.

Name	Description
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_socket	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_socket` class template provides functionality that is common to both stream-oriented and datagram-oriented sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.33 basic_raw_socket::max_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.71.34 basic_raw_socket::message_do_not_route

Inherited from socket_base.

Specify that the data should not be subject to routing.

```
static const int message_do_not_route = implementation_defined;
```

5.71.35 basic_raw_socket::message_end_of_record

Inherited from socket_base.

Specifies that the data marks the end of a record.

```
static const int message_end_of_record = implementation_defined;
```

5.71.36 basic_raw_socket::message_flags

Inherited from socket_base.

Bitmask type for flags that can be passed to send and receive operations.

```
typedef int message_flags;
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.37 basic_raw_socket::message_out_of_band

Inherited from socket_base.

Process out-of-band data.

```
static const int message_out_of_band = implementation_defined;
```

5.71.38 basic_raw_socket::message_peek

Inherited from socket_base.

Peek at incoming data without removing it from the input queue.

```
static const int message_peek = implementation_defined;
```

5.71.39 basic_raw_socket::native

Inherited from basic_socket.

(Deprecated: Use native_handle().) Get the native socket representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.71.40 basic_raw_socket::native_handle

Inherited from basic_socket.

Get the native socket representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.71.41 basic_raw_socket::native_handle_type

The native representation of a socket.

```
typedef RawSocketService::native_handle_type native_handle_type;
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.42 basic_raw_socket::native_non_blocking

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.71.42.1 basic_raw_socket::native_non_blocking (1 of 3 overloads)

Inherited from `basic_socket`.

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

This function is used to retrieve the non-blocking mode of the underlying native socket. This mode has no effect on the behaviour of the socket object's synchronous operations.

Return Value

`true` if the underlying socket is in non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Remarks

The current non-blocking mode is cached by the socket object. Consequently, the return value may be incorrect if the non-blocking mode was set directly on the native socket.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.
                errno = 0;
                int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
                ec = asio::error_code(n < 0 ? errno : 0,
                                     asio::error::get_system_category());
                total_bytes_transferred_ += ec ? 0 : n;

                // Retry operation immediately if interrupted by signal.
                if (ec == asio::error::interrupted)
                    continue;
            }
        }
    }
};
```

```

// Check if we need to run the operation again.
if (ec == asio::error::would_block
    || ec == asio::error::try_again)
{
    // We have to wait for the socket to become ready again.
    sock_.async_write_some(asio::null_buffers(), *this);
    return;
}

if (ec || n == 0)
{
    // An error occurred, or we have reached the end of the file.
    // Either way we must exit the loop so we can call the handler.
    break;
}

// Loop around to try calling sendfile again.
}
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}
};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.71.42.2 basic_raw_socket::native_non_blocking (2 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);
```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Exceptions

asio::system_error Thrown on failure. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.
                errno = 0;
                int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
                ec = asio::error_code(n < 0 ? errno : 0,
                                      asio::error::get_system_category());
                total_bytes_transferred_ += ec ? 0 : n;

                // Retry operation immediately if interrupted by signal.
                if (ec == asio::error::interrupted)
                    continue;

                // Check if we need to run the operation again.
                if (ec == asio::error::would_block
                    || ec == asio::error::try_again)
                {
                    // We have to wait for the socket to become ready again.
                    sock_.async_write_some(asio::null_buffers(), *this);
                    return;
                }

                if (ec || n == 0)
                {
                    // An error occurred, or we have reached the end of the file.
                    // Either way we must exit the loop so we can call the handler.
                    break;
                }
            }

            // Loop around to try calling sendfile again.
        }

        // Pass result back to user's handler.
        handler_(ec, total_bytes_transferred_);
    }
}
```

```

};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.71.42.3 basic_raw_socket::native_non_blocking (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);

```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

ec Set to indicate what error occurred, if any. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```

template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.

```

```

errno = 0;
int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
ec = asio::error_code(n < 0 ? errno : 0,
    asio::error::get_system_category());
total_bytes_transferred_ += ec ? 0 : n;

// Retry operation immediately if interrupted by signal.
if (ec == asio::error::interrupted)
    continue;

// Check if we need to run the operation again.
if (ec == asio::error::would_block
    || ec == asio::error::try_again)
{
    // We have to wait for the socket to become ready again.
    sock_.async_write_some(asio::null_buffers(), *this);
    return;
}

if (ec || n == 0)
{
    // An error occurred, or we have reached the end of the file.
    // Either way we must exit the loop so we can call the handler.
    break;
}

// Loop around to try calling sendfile again.
}
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}
};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.71.43 basic_raw_socket::native_type

(Deprecated: Use native_handle_type.) The native representation of a socket.

```
typedef RawSocketService::native_handle_type native_type;
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.44 basic_raw_socket::non_blocking

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Sets the non-blocking mode of the socket.

```
void non_blocking(
    bool mode);

asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.71.44.1 basic_raw_socket::non_blocking (1 of 3 overloads)

Inherited from basic_socket.

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Return Value

`true` if the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.71.44.2 basic_raw_socket::non_blocking (2 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
void non_blocking(
    bool mode);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.71.44.3 basic_raw_socket::non_blocking (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

ec Set to indicate what error occurred, if any.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.71.45 basic_raw_socket::non_blocking_io

Inherited from socket_base.

(Deprecated: Use `non_blocking()` IO control command to set the blocking mode of the socket.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::non_blocking_io command(true);
socket.io_control(command);
```

Requirements

Header: `asio/basic_raw_socket.hpp`

Convenience header: `asio.hpp`

5.71.46 basic_raw_socket::open

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());

asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.71.46.1 basic_raw_socket::open (1 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying protocol parameters to be used.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
```

5.71.46.2 basic_raw_socket::open (2 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying which protocol is to be used.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.71.47 basic_raw_socket::operator=

Move-assign a `basic_raw_socket` from another.

```
basic_raw_socket & operator=(  
    basic_raw_socket && other);
```

Move-assign a `basic_raw_socket` from a socket of another protocol type.

```
template<  
    typename Protocol,  
    typename RawSocketService1>  
enable_if< is_convertible< Protocol, Protocol >::value, basic_raw_socket >::type & operator=(  
    basic_raw_socket< Protocol, RawSocketService1 > && other);
```

5.71.47.1 basic_raw_socket::operator= (1 of 2 overloads)

Move-assign a `basic_raw_socket` from another.

```
basic_raw_socket & operator=(  
    basic_raw_socket && other);
```

This assignment operator moves a raw socket from one object to another.

Parameters

other The other `basic_raw_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_raw_socket (io_service&)` constructor.

5.71.47.2 basic_raw_socket::operator= (2 of 2 overloads)

Move-assign a `basic_raw_socket` from a socket of another protocol type.

```
template<  
    typename Protocol,  
    typename RawSocketService1>  
enable_if< is_convertible< Protocol, Protocol >::value, basic_raw_socket >::type & operator=(  
    basic_raw_socket< Protocol, RawSocketService1 > && other);
```

This assignment operator moves a raw socket from one object to another.

Parameters

other The other `basic_raw_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_raw_socket (io_service&)` constructor.

5.71.48 basic_raw_socket::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.49 basic_raw_socket::receive

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers);

template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags);

template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.71.49.1 basic_raw_socket::receive (1 of 3 overloads)

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers);
```

This function is used to receive data on the raw socket. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

Return Value

The number of bytes received.

Exceptions

asio::system_error Thrown on failure.

Remarks

The receive operation can only be used with a connected socket. Use the receive_from function to receive data on an unconnected raw socket.

Example

To receive into a single data buffer use the **buffer** function as follows:

```
socket.receive(asio::buffer(data, size));
```

See the **buffer** documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

5.71.49.2 basic_raw_socket::receive (2 of 3 overloads)

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags);
```

This function is used to receive data on the raw socket. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

flags Flags specifying how the receive call is to be made.

Return Value

The number of bytes received.

Exceptions

asio::system_error Thrown on failure.

Remarks

The receive operation can only be used with a connected socket. Use the receive_from function to receive data on an unconnected raw socket.

5.71.49.3 basic_raw_socket::receive (3 of 3 overloads)

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to receive data on the raw socket. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

flags Flags specifying how the receive call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes received.

Remarks

The receive operation can only be used with a connected socket. Use the receive_from function to receive data on an unconnected raw socket.

5.71.50 basic_raw_socket::receive_buffer_size

Inherited from socket_base.

Socket option for the receive buffer size of a socket.

```
typedef implementation_defined receive_buffer_size;
```

Implements the SOL_SOCKET/SO_RCVBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.51 basic_raw_socket::receive_from

Receive raw data with the endpoint of the sender.

```
template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint);

template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags);

template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.71.51.1 basic_raw_socket::receive_from (1 of 3 overloads)

Receive raw data with the endpoint of the sender.

```
template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint);
```

This function is used to receive raw data. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

sender_endpoint An endpoint object that receives the endpoint of the remote sender of the data.

Return Value

The number of bytes received.

Exceptions

asio::system_error Thrown on failure.

Example

To receive into a single data buffer use the **buffer** function as follows:

```
asio::ip::udp::endpoint sender_endpoint;
socket.receive_from(
    asio::buffer(data, size), sender_endpoint);
```

See the **buffer** documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.71.51.2 basic_raw_socket::receive_from (2 of 3 overloads)

Receive raw data with the endpoint of the sender.

```
template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags);
```

This function is used to receive raw data. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

sender_endpoint An endpoint object that receives the endpoint of the remote sender of the data.

flags Flags specifying how the receive call is to be made.

Return Value

The number of bytes received.

Exceptions

asio::system_error Thrown on failure.

5.71.51.3 basic_raw_socket::receive_from (3 of 3 overloads)

Receive raw data with the endpoint of the sender.

```
template<
    typename MutableBufferSequence>
std::size_t receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to receive raw data. The function call will block until data has been received successfully or an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

sender_endpoint An endpoint object that receives the endpoint of the remote sender of the data.

flags Flags specifying how the receive call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes received.

5.71.52 basic_raw_socket::receive_low_watermark

Inherited from socket_base.

Socket option for the receive low watermark.

```
typedef implementation_defined receive_low_watermark;
```

Implements the SOL_SOCKET/SO_RCVLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.53 basic_raw_socket::remote_endpoint

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;

endpoint_type remote_endpoint(
    asio::error_code & ec) const;
```

5.71.53.1 basic_raw_socket::remote_endpoint (1 of 2 overloads)

Inherited from basic_socket.

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;
```

This function is used to obtain the remote endpoint of the socket.

Return Value

An object that represents the remote endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint();
```

5.71.53.2 basic_raw_socket::remote_endpoint (2 of 2 overloads)

Inherited from basic_socket.

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint(
    asio::error_code & ec) const;
```

This function is used to obtain the remote endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the remote endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.71.54 basic_raw_socket::reuse_address

Inherited from socket_base.

Socket option to allow the socket to be bound to an address that is already in use.

```
typedef implementation_defined reuse_address;
```

Implements the SOL_SOCKET/SO_REUSEADDR socket option.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.55 basic_raw_socket::send

Send some data on a connected socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags);

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.71.55.1 basic_raw_socket::send (1 of 3 overloads)

Send some data on a connected socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers);
```

This function is used to send data on the raw socket. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One ore more data buffers to be sent on the socket.

Return Value

The number of bytes sent.

Exceptions

asio::system_error Thrown on failure.

Remarks

The send operation can only be used with a connected socket. Use the send_to function to send data on an unconnected raw socket.

Example

To send a single data buffer use the **buffer** function as follows:

```
socket.send(asio::buffer(data, size));
```

See the **buffer** documentation for information on sending multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

5.71.55.2 basic_raw_socket::send (2 of 3 overloads)

Send some data on a connected socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags);
```

This function is used to send data on the raw socket. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One ore more data buffers to be sent on the socket.

flags Flags specifying how the send call is to be made.

Return Value

The number of bytes sent.

Exceptions

asio::system_error Thrown on failure.

Remarks

The send operation can only be used with a connected socket. Use the `send_to` function to send data on an unconnected raw socket.

5.71.55.3 `basic_raw_socket::send` (3 of 3 overloads)

Send some data on a connected socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to send data on the raw socket. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One or more data buffers to be sent on the socket.

flags Flags specifying how the send call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes sent.

Remarks

The send operation can only be used with a connected socket. Use the `send_to` function to send data on an unconnected raw socket.

5.71.56 `basic_raw_socket::send_buffer_size`

Inherited from `socket_base`.

Socket option for the send buffer size of a socket.

```
typedef implementation_defined send_buffer_size;
```

Implements the SOL_SOCKET/SO_SNDBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.57 basic_raw_socket::send_low_watermark

Inherited from socket_base.

Socket option for the send low watermark.

```
typedef implementation_defined send_low_watermark;
```

Implements the SOL_SOCKET/SO SNDLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.58 basic_raw_socket::send_to

Send raw data to the specified endpoint.

```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination);
```



```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags);
```



```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.71.58.1 basic_raw_socket::send_to (1 of 3 overloads)

Send raw data to the specified endpoint.

```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination);
```

This function is used to send raw data to the specified remote endpoint. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One or more data buffers to be sent to the remote endpoint.

destination The remote endpoint to which the data will be sent.

Return Value

The number of bytes sent.

Exceptions

asio::system_error Thrown on failure.

Example

To send a single data buffer use the **buffer** function as follows:

```
asio::ip::udp::endpoint destination(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.send_to(asio::buffer(data, size), destination);
```

See the **buffer** documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.71.58.2 basic_raw_socket::send_to (2 of 3 overloads)

Send raw data to the specified endpoint.

```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags);
```

This function is used to send raw data to the specified remote endpoint. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One or more data buffers to be sent to the remote endpoint.

destination The remote endpoint to which the data will be sent.

flags Flags specifying how the send call is to be made.

Return Value

The number of bytes sent.

Exceptions

`asio::system_error` Thrown on failure.

5.71.58.3 basic_raw_socket::send_to (3 of 3 overloads)

Send raw data to the specified endpoint.

```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to send raw data to the specified remote endpoint. The function call will block until the data has been sent successfully or an error occurs.

Parameters

buffers One or more data buffers to be sent to the remote endpoint.

destination The remote endpoint to which the data will be sent.

flags Flags specifying how the send call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes sent.

5.71.59 basic_raw_socket::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`.) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.71.60 basic_raw_socket::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef RawSocketService service_type;
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.61 basic_raw_socket::set_option

Set an option on the socket.

```
void set_option(
    const SettableSocketOption & option);

asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.71.61.1 basic_raw_socket::set_option (1 of 2 overloads)

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.71.61.2 basic_raw_socket::set_option (2 of 2 overloads)

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

ec Set to indicate what error occurred, if any.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.71.62 basic_raw_socket::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.71.62.1 basic_raw_socket::shutdown (1 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

Exceptions

asio::system_error Thrown on failure.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
socket.shutdown(asio::ip::tcp::socket::shutdown_send);
```

5.71.62.2 basic_raw_socket::shutdown (2 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

ec Set to indicate what error occurred, if any.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);
if (ec)
{
    // An error occurred.
}
```

5.71.63 basic_raw_socket::shutdown_type

Inherited from socket_base.

Different ways a socket may be shutdown.

```
enum shutdown_type
```

Values

shutdown_receive Shutdown the receive side of the socket.

shutdown_send Shutdown the send side of the socket.

shutdown_both Shutdown both send and receive on the socket.

5.72 basic_seq_packet_socket

Provides sequenced packet socket functionality.

```
template<
    typename Protocol,
    typename SeqPacketSocketService = seq_packet_socket_service<Protocol>>
class basic_seq_packet_socket :
    public basic_socket< Protocol, SeqPacketSocketService >
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive.
async_send	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_seq_packet_socket	Construct a basic_seq_packet_socket without opening it. Construct and open a basic_seq_packet_socket. Construct a basic_seq_packet_socket, opening it and binding it to the given local endpoint. Construct a basic_seq_packet_socket on an existing native socket. Move-construct a basic_seq_packet_socket from another. Move-construct a basic_seq_packet_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.

Name	Description
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.
operator=	Move-assign a basic_seq_packet_socket from another. Move-assign a basic_seq_packet_socket from a socket of another protocol type.
receive	Receive some data on the socket. Receive some data on a connected socket.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on the socket.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_seq_packet_socket` class template provides asynchronous and blocking sequenced packet socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/basic_seq_packet_socket.hpp`

Convenience header: `asio.hpp`

5.72.1 `basic_seq_packet_socket::assign`

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);

asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.72.1.1 `basic_seq_packet_socket::assign (1 of 2 overloads)`

Inherited from basic_socket.

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

5.72.1.2 `basic_seq_packet_socket::assign (2 of 2 overloads)`

Inherited from basic_socket.

Assign an existing native socket to the socket.

```
asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.72.2 basic_seq_packet_socket::async_connect

Inherited from `basic_socket`.

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
void-or-deduced async_connect(
    const endpoint_type & peer_endpoint,
    ConnectHandler handler);
```

This function is used to asynchronously connect a socket to the specified remote endpoint. The function call always returns immediately.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected. Copies will be made of the endpoint object as required.

handler The handler to be called when the connection operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error // Result of operation
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

```
void connect_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Connect succeeded.
    }
}

...

asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.async_connect(endpoint, connect_handler);
```

5.72.3 basic_seq_packet_socket::async_receive

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
```

```

socket_base::message_flags & out_flags,
ReadHandler handler);

template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags in_flags,
    socket_base::message_flags & out_flags,
    ReadHandler handler);

```

5.72.3.1 basic_seq_packet_socket::async_receive (1 of 2 overloads)

Start an asynchronous receive.

```

template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags & out_flags,
    ReadHandler handler);

```

This function is used to asynchronously receive data from the sequenced packet socket. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

out_flags Once the asynchronous operation completes, contains flags associated with the received data. For example, if the `socket_base::message_end_of_record` bit is set then the received data marks the end of a record. The caller must guarantee that the referenced variable remains valid until the handler is called.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To receive into a single data buffer use the **buffer** function as follows:

```
socket.async_receive(asio::buffer(data, size), out_flags, handler);
```

See the **buffer** documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.72.3.2 basic_seq_packet_socket::async_receive (2 of 2 overloads)

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags in_flags,
    socket_base::message_flags & out_flags,
    ReadHandler handler);
```

This function is used to asynchronously receive data from the sequenced data socket. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

in_flags Flags specifying how the receive call is to be made.

out_flags Once the asynchronous operation completes, contains flags associated with the received data. For example, if the `socket_base::message_end_of_record` bit is set then the received data marks the end of a record. The caller must guarantee that the referenced variable remains valid until the handler is called.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To receive into a single data buffer use the **buffer** function as follows:

```
socket.async_receive(
    asio::buffer(data, size),
    0, out_flags, handler);
```

See the **buffer** documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.72.4 basic_seq_packet_socket::async_send

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);
```

This function is used to asynchronously send data on the sequenced packet socket. The function call always returns immediately.

Parameters

buffers One or more data buffers to be sent on the socket. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

flags Flags specifying how the send call is to be made.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(  
    const asio::error_code& error, // Result of operation.  
    std::size_t bytes_transferred           // Number of bytes sent.  
) ;
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

To send a single data buffer use the `buffer` function as follows:

```
socket.async_send(asio::buffer(data, size), 0, handler);
```

See the `buffer` documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.72.5 basic_seq_packet_socket::at_mark

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;  
  
bool at_mark(  
    asio::error_code & ec) const;
```

5.72.5.1 basic_seq_packet_socket::at_mark (1 of 2 overloads)

Inherited from basic_socket.

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

Exceptions

`asio::system_error` Thrown on failure.

5.72.5.2 `basic_seq_packet_socket::at_mark` (2 of 2 overloads)

Inherited from basic_socket.

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark(  
    asio::error_code & ec) const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

5.72.6 `basic_seq_packet_socket::available`

Determine the number of bytes available for reading.

```
std::size_t available() const;  
  
std::size_t available(  
    asio::error_code & ec) const;
```

5.72.6.1 `basic_seq_packet_socket::available` (1 of 2 overloads)

Inherited from basic_socket.

Determine the number of bytes available for reading.

```
std::size_t available() const;
```

This function is used to determine the number of bytes that may be read without blocking.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

Exceptions

`asio::system_error` Thrown on failure.

5.72.6.2 `basic_seq_packet_socket::available` (2 of 2 overloads)

Inherited from basic_socket.

Determine the number of bytes available for reading.

```
std::size_t available(  
    asio::error_code & ec) const;
```

This function is used to determine the number of bytes that may be read without blocking.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

5.72.7 basic_seq_packet_socket::basic_seq_packet_socket

Construct a `basic_seq_packet_socket` without opening it.

```
explicit basic_seq_packet_socket(
   asio::io_service & io_service);
```

Construct and open a `basic_seq_packet_socket`.

```
basic_seq_packet_socket(
    asio::io_service & io_service,
    const protocol_type & protocol);
```

Construct a `basic_seq_packet_socket`, opening it and binding it to the given local endpoint.

```
basic_seq_packet_socket(
    asio::io_service & io_service,
    const endpoint_type & endpoint);
```

Construct a `basic_seq_packet_socket` on an existing native socket.

```
basic_seq_packet_socket(
    asio::io_service & io_service,
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

Move-construct a `basic_seq_packet_socket` from another.

```
basic_seq_packet_socket(
    basic_seq_packet_socket && other);
```

Move-construct a `basic_seq_packet_socket` from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename SeqPacketSocketService1>
basic_seq_packet_socket(
    basic_seq_packet_socket< Protocol1, SeqPacketSocketService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.72.7.1 basic_seq_packet_socket::basic_seq_packet_socket (1 of 6 overloads)

Construct a `basic_seq_packet_socket` without opening it.

```
basic_seq_packet_socket(
    asio::io_service & io_service);
```

This constructor creates a sequenced packet socket without opening it. The socket needs to be opened and then connected or accepted before data can be sent or received on it.

Parameters

io_service The `io_service` object that the sequenced packet socket will use to dispatch handlers for any asynchronous operations performed on the socket.

5.72.7.2 basic_seq_packet_socket::basic_seq_packet_socket (2 of 6 overloads)

Construct and open a `basic_seq_packet_socket`.

```
basic_seq_packet_socket(
    asio::io_service & io_service,
    const protocol_type & protocol);
```

This constructor creates and opens a sequenced_packet socket. The socket needs to be connected or accepted before data can be sent or received on it.

Parameters

io_service The `io_service` object that the sequenced packet socket will use to dispatch handlers for any asynchronous operations performed on the socket.

protocol An object specifying protocol parameters to be used.

Exceptions

`asio::system_error` Thrown on failure.

5.72.7.3 basic_seq_packet_socket::basic_seq_packet_socket (3 of 6 overloads)

Construct a `basic_seq_packet_socket`, opening it and binding it to the given local endpoint.

```
basic_seq_packet_socket(
    asio::io_service & io_service,
    const endpoint_type & endpoint);
```

This constructor creates a sequenced packet socket and automatically opens it bound to the specified endpoint on the local machine. The protocol used is the protocol associated with the given endpoint.

Parameters

io_service The `io_service` object that the sequenced packet socket will use to dispatch handlers for any asynchronous operations performed on the socket.

endpoint An endpoint on the local machine to which the sequenced packet socket will be bound.

Exceptions

asio::system_error Thrown on failure.

5.72.7.4 basic_seq_packet_socket::basic_seq_packet_socket (4 of 6 overloads)

Construct a **basic_seq_packet_socket** on an existing native socket.

```
basic_seq_packet_socket(
    asio::io_service & io_service,
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

This constructor creates a sequenced packet socket object to hold an existing native socket.

Parameters

io_service The **io_service** object that the sequenced packet socket will use to dispatch handlers for any asynchronous operations performed on the socket.

protocol An object specifying protocol parameters to be used.

native_socket The new underlying socket implementation.

Exceptions

asio::system_error Thrown on failure.

5.72.7.5 basic_seq_packet_socket::basic_seq_packet_socket (5 of 6 overloads)

Move-construct a **basic_seq_packet_socket** from another.

```
basic_seq_packet_socket(
    basic_seq_packet_socket && other);
```

This constructor moves a sequenced packet socket from one object to another.

Parameters

other The other **basic_seq_packet_socket** object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_seq_packet_socket(io_service&)` constructor.

5.72.7.6 basic_seq_packet_socket::basic_seq_packet_socket (6 of 6 overloads)

Move-construct a **basic_seq_packet_socket** from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename SeqPacketSocketService1>
basic_seq_packet_socket(
    basic_seq_packet_socket< Protocol1, SeqPacketSocketService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

This constructor moves a sequenced packet socket from one object to another.

Parameters

other The other `basic_seq_packet_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_seq_packet_socket(io_service&)` constructor.

5.72.8 `basic_seq_packet_socket::bind`

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.72.8.1 `basic_seq_packet_socket::bind (1 of 2 overloads)`

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

Exceptions

`asio::system_error` Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345));
```

5.72.8.2 basic_seq_packet_socket::bind (2 of 2 overloads)

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
asio::error_code ec;
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345), ec);
if (ec)
{
    // An error occurred.
}
```

5.72.9 basic_seq_packet_socket::broadcast

Inherited from socket_base.

Socket option to permit sending of broadcast messages.

```
typedef implementation_defined broadcast;
```

Implements the SOL_SOCKET/SO_BROADCAST socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.10 basic_seq_packet_socket::bytes_readable

Inherited from socket_base.

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::bytes_readable command(true);
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.11 basic_seq_packet_socket::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.72.11.1 basic_seq_packet_socket::cancel (1 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
void cancel();
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the asio::error::operation_aborted error.

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.72.11.2 `basic_seq_packet_socket::cancel (2 of 2 overloads)`

Inherited from `basic_socket`.

Cancel all asynchronous operations associated with the socket.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.72.12 basic_seq_packet_socket::close

Close the socket.

```
void close();  
  
asio::error_code close(  
    asio::error_code & ec);
```

5.72.12.1 basic_seq_packet_socket::close (1 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
void close();
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure. Note that, even if the function indicates an error, the underlying descriptor is closed.

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.72.12.2 basic_seq_packet_socket::close (2 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
asio::error_code close(  
    asio::error_code & ec);
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any. Note that, even if the function indicates an error, the underlying descriptor is closed.

Example

```
asio::ip::tcp::socket socket(io_service);  
...  
asio::error_code ec;  
socket.close(ec);  
if (ec)  
{  
    // An error occurred.  
}
```

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.72.13 `basic_seq_packet_socket::connect`

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);

asio::error_code connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.72.13.1 `basic_seq_packet_socket::connect (1 of 2 overloads)`

Inherited from basic_socket.

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.connect(endpoint);
```

5.72.13.2 basic_seq_packet_socket::connect (2 of 2 overloads)

Inherited from basic_socket.

Connect the socket to the specified endpoint.

```
asio::error_code connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
asio::error_code ec;
socket.connect(endpoint, ec);
if (ec)
{
    // An error occurred.
}
```

5.72.14 basic_seq_packet_socket::debug

Inherited from socket_base.

Socket option to enable socket-level debugging.

```
typedef implementation_defined debug;
```

Implements the SOL_SOCKET/SO_DEBUG socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.15 basic_seq_packet_socket::do_not_route

Inherited from socket_base.

Socket option to prevent routing, use local interfaces only.

```
typedef implementation_defined do_not_route;
```

Implements the SOL_SOCKET/SO_DONTROUTE socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.16 basic_seq_packet_socket::enable_connection_aborted

Inherited from socket_base.

Socket option to report aborted connections on accept.

```
typedef implementation_defined enable_connection_aborted;
```

Implements a custom socket option that determines whether or not an accept operation is permitted to fail with `asio::error::connection_aborted`. By default the option is false.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.17 basic_seq_packet_socket::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.18 basic_seq_packet_socket::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();

const implementation_type & get_implementation() const;
```

5.72.18.1 basic_seq_packet_socket::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.72.18.2 `basic_seq_packet_socket::get_implementation` (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.72.19 `basic_seq_packet_socket::get_io_service`

Inherited from basic_io_object.

Get the `io_service` associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the `io_service` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_service` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.72.20 `basic_seq_packet_socket::get_option`

Get an option from the socket.

```
void get_option(
    GettableSocketOption & option) const;

asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.72.20.1 `basic_seq_packet_socket::get_option` (1 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

Exceptions

`asio::system_error` Thrown on failure.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.72.20.2 basic_seq_packet_socket::get_option (2 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
asio::error_code ec;
socket.get_option(option, ec);
if (ec)
{
    // An error occurred.
}
bool is_set = option.value();
```

5.72.21 basic_seq_packet_socket::get_service

Get the service associated with the I/O object.

```
service_type & get_service();

const service_type & get_service() const;
```

5.72.21.1 basic_seq_packet_socket::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.72.21.2 basic_seq_packet_socket::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.72.22 basic_seq_packet_socket::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.72.23 basic_seq_packet_socket::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.24 basic_seq_packet_socket::io_control

Perform an IO control command on the socket.

```
void io_control(
    IoControlCommand & command);

asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.72.24.1 basic_seq_packet_socket::io_control (1 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

5.72.24.2 basic_seq_packet_socket::io_control (2 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
asio::error_code ec;
socket.io_control(command, ec);
if (ec)
{
    // An error occurred.
}
std::size_t bytes_readable = command.get();
```

5.72.25 basic_seq_packet_socket::is_open

Inherited from basic_socket.

Determine whether the socket is open.

```
bool is_open() const;
```

5.72.26 basic_seq_packet_socket::keep_alive

Inherited from socket_base.

Socket option to send keep-alives.

```
typedef implementation_defined keep_alive;
```

Implements the SOL_SOCKET/SO_KEEPALIVE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.27 basic_seq_packet_socket::linger

Inherited from socket_base.

Socket option to specify whether the socket lingers on close if unsent data is present.

```
typedef implementation_defined linger;
```

Implements the SOL_SOCKET/SO_LINGER socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option;
socket.get_option(option);
bool is_set = option.enabled();
unsigned short timeout = option.timeout();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.28 basic_seq_packet_socket::local_endpoint

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;

endpoint_type local_endpoint(
    asio::error_code & ec) const;
```

5.72.28.1 basic_seq_packet_socket::local_endpoint (1 of 2 overloads)

Inherited from basic_socket.

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;
```

This function is used to obtain the locally bound endpoint of the socket.

Return Value

An object that represents the local endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::endpoint endpoint = socket.local_endpoint();
```

5.72.28.2 basic_seq_packet_socket::local_endpoint (2 of 2 overloads)

Inherited from basic_socket.

Get the local endpoint of the socket.

```
endpoint_type local_endpoint(
    asio::error_code & ec) const;
```

This function is used to obtain the locally bound endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the local endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.local_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.72.29 basic_seq_packet_socket::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.72.29.1 basic_seq_packet_socket::lowest_layer (1 of 2 overloads)

Inherited from basic_socket.

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a **basic_socket** cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.72.29.2 basic_seq_packet_socket::lowest_layer (2 of 2 overloads)

Inherited from basic_socket.

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a **basic_socket** cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.72.30 basic_seq_packet_socket::lowest_layer_type

Inherited from basic_socket.

A **basic_socket** is always the lowest layer.

```
typedef basic_socket< Protocol, SeqPacketSocketService > lowest_layer_type;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.

Name	Description
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.

Name	Description
<code>basic_socket</code>	Construct a <code>basic_socket</code> without opening it. Construct and open a <code>basic_socket</code> . Construct a <code>basic_socket</code> , opening it and binding it to the given local endpoint. Construct a <code>basic_socket</code> on an existing native socket. Move-construct a <code>basic_socket</code> from another. Move-construct a <code>basic_socket</code> from a socket of another protocol type.
<code>bind</code>	Bind the socket to the given local endpoint.
<code>cancel</code>	Cancel all asynchronous operations associated with the socket.
<code>close</code>	Close the socket.
<code>connect</code>	Connect the socket to the specified endpoint.
<code>get_io_service</code>	Get the <code>io_service</code> associated with the object.
<code>get_option</code>	Get an option from the socket.
<code>io_control</code>	Perform an IO control command on the socket.
<code>is_open</code>	Determine whether the socket is open.
<code>local_endpoint</code>	Get the local endpoint of the socket.
<code>lowest_layer</code>	Get a reference to the lowest layer. Get a const reference to the lowest layer.
<code>native</code>	(Deprecated: Use <code>native_handle()</code>) Get the native socket representation.
<code>native_handle</code>	Get the native socket representation.
<code>native_non_blocking</code>	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
<code>non_blocking</code>	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
<code>open</code>	Open the socket using the specified protocol.
<code>operator=</code>	Move-assign a <code>basic_socket</code> from another. Move-assign a <code>basic_socket</code> from a socket of another protocol type.
<code>remote_endpoint</code>	Get the remote endpoint of the socket.
<code>set_option</code>	Set an option on the socket.

Name	Description
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_socket	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_socket` class template provides functionality that is common to both stream-oriented and datagram-oriented sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.31 basic_seq_packet_socket::max_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.72.32 basic_seq_packet_socket::message_do_not_route

Inherited from socket_base.

Specify that the data should not be subject to routing.

```
static const int message_do_not_route = implementation_defined;
```

5.72.33 basic_seq_packet_socket::message_end_of_record

Inherited from socket_base.

Specifies that the data marks the end of a record.

```
static const int message_end_of_record = implementation_defined;
```

5.72.34 basic_seq_packet_socket::message_flags

Inherited from socket_base.

Bitmask type for flags that can be passed to send and receive operations.

```
typedef int message_flags;
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.35 basic_seq_packet_socket::message_out_of_band

Inherited from socket_base.

Process out-of-band data.

```
static const int message_out_of_band = implementation_defined;
```

5.72.36 basic_seq_packet_socket::message_peek

Inherited from socket_base.

Peek at incoming data without removing it from the input queue.

```
static const int message_peek = implementation_defined;
```

5.72.37 basic_seq_packet_socket::native

Inherited from basic_socket.

(Deprecated: Use native_handle().) Get the native socket representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.72.38 basic_seq_packet_socket::native_handle

Inherited from basic_socket.

Get the native socket representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.72.39 basic_seq_packet_socket::native_handle_type

The native representation of a socket.

```
typedef SeqPacketSocketService::native_handle_type native_handle_type;
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.40 basic_seq_packet_socket::native_non_blocking

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.72.40.1 basic_seq_packet_socket::native_non_blocking (1 of 3 overloads)

Inherited from `basic_socket`.

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

This function is used to retrieve the non-blocking mode of the underlying native socket. This mode has no effect on the behaviour of the socket object's synchronous operations.

Return Value

`true` if the underlying socket is in non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Remarks

The current non-blocking mode is cached by the socket object. Consequently, the return value may be incorrect if the non-blocking mode was set directly on the native socket.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.
                errno = 0;
                int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
                ec = asio::error_code(n < 0 ? errno : 0,
                                     asio::error::get_system_category());
                total_bytes_transferred_ += ec ? 0 : n;

                // Retry operation immediately if interrupted by signal.
                if (ec == asio::error::interrupted)
                    continue;
            }
        }
    }
};
```

```

// Check if we need to run the operation again.
if (ec == asio::error::would_block
    || ec == asio::error::try_again)
{
    // We have to wait for the socket to become ready again.
    sock_.async_write_some(asio::null_buffers(), *this);
    return;
}

if (ec || n == 0)
{
    // An error occurred, or we have reached the end of the file.
    // Either way we must exit the loop so we can call the handler.
    break;
}

// Loop around to try calling sendfile again.
}
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}
};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.72.40.2 basic_seq_packet_socket::native_non_blocking (2 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);
```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Exceptions

asio::system_error Thrown on failure. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.
                errno = 0;
                int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
                ec = asio::error_code(n < 0 ? errno : 0,
                                      asio::error::get_system_category());
                total_bytes_transferred_ += ec ? 0 : n;

                // Retry operation immediately if interrupted by signal.
                if (ec == asio::error::interrupted)
                    continue;

                // Check if we need to run the operation again.
                if (ec == asio::error::would_block
                    || ec == asio::error::try_again)
                {
                    // We have to wait for the socket to become ready again.
                    sock_.async_write_some(asio::null_buffers(), *this);
                    return;
                }

                if (ec || n == 0)
                {
                    // An error occurred, or we have reached the end of the file.
                    // Either way we must exit the loop so we can call the handler.
                    break;
                }
            }

            // Loop around to try calling sendfile again.
        }

        // Pass result back to user's handler.
        handler_(ec, total_bytes_transferred_);
    }
}
```

```

};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.72.40.3 basic_seq_packet_socket::native_non_blocking (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);

```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

ec Set to indicate what error occurred, if any. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```

template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.

```

```

errno = 0;
int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
ec = asio::error_code(n < 0 ? errno : 0,
    asio::error::get_system_category());
total_bytes_transferred_ += ec ? 0 : n;

// Retry operation immediately if interrupted by signal.
if (ec == asio::error::interrupted)
    continue;

// Check if we need to run the operation again.
if (ec == asio::error::would_block
    || ec == asio::error::try_again)
{
    // We have to wait for the socket to become ready again.
    sock_.async_write_some(asio::null_buffers(), *this);
    return;
}

if (ec || n == 0)
{
    // An error occurred, or we have reached the end of the file.
    // Either way we must exit the loop so we can call the handler.
    break;
}

// Loop around to try calling sendfile again.
}
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}
};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.72.41 basic_seq_packet_socket::native_type

(Deprecated: Use native_handle_type.) The native representation of a socket.

```
typedef SeqPacketSocketService::native_handle_type native_type;
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.42 basic_seq_packet_socket::non_blocking

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Sets the non-blocking mode of the socket.

```
void non_blocking(
    bool mode);

asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.72.42.1 basic_seq_packet_socket::non_blocking (1 of 3 overloads)

Inherited from basic_socket.

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Return Value

`true` if the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.72.42.2 basic_seq_packet_socket::non_blocking (2 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
void non_blocking(
    bool mode);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.72.42.3 basic_seq_packet_socket::non_blocking (3 of 3 overloads)

Inherited from `basic_socket`.

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

ec Set to indicate what error occurred, if any.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.72.43 basic_seq_packet_socket::non_blocking_io

Inherited from `socket_base`.

(Deprecated: Use `non_blocking()` IO control command to set the blocking mode of the socket.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::non_blocking_io command(true);
socket.io_control(command);
```

Requirements

Header: `asio/basic_seq_packet_socket.hpp`

Convenience header: `asio.hpp`

5.72.44 basic_seq_packet_socket::open

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());

asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.72.44.1 basic_seq_packet_socket::open (1 of 2 overloads)

Inherited from `basic_socket`.

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying protocol parameters to be used.

Exceptions

`asio::system_error` Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
```

5.72.44.2 basic_seq_packet_socket::open (2 of 2 overloads)

Inherited from `basic_socket`.

Open the socket using the specified protocol.

```
asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying which protocol is to be used.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.72.45 basic_seq_packet_socket::operator=

Move-assign a `basic_seq_packet_socket` from another.

```
basic_seq_packet_socket & operator=(  
    basic_seq_packet_socket && other);
```

Move-assign a `basic_seq_packet_socket` from a socket of another protocol type.

```
template<  
    typename Protocol,  
    typename SeqPacketSocketService1>  
enable_if< is_convertible< Protocol, Protocol >::value, basic_seq_packet_socket >::type & ←  
operator=(  
    basic_seq_packet_socket< Protocol, SeqPacketSocketService1 > && other);
```

5.72.45.1 basic_seq_packet_socket::operator= (1 of 2 overloads)

Move-assign a `basic_seq_packet_socket` from another.

```
basic_seq_packet_socket & operator=(  
    basic_seq_packet_socket && other);
```

This assignment operator moves a sequenced packet socket from one object to another.

Parameters

other The other `basic_seq_packet_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_seq_packet_socket(io_service&)` constructor.

5.72.45.2 basic_seq_packet_socket::operator= (2 of 2 overloads)

Move-assign a `basic_seq_packet_socket` from a socket of another protocol type.

```
template<  
    typename Protocol,  
    typename SeqPacketSocketService1>  
enable_if< is_convertible< Protocol, Protocol >::value, basic_seq_packet_socket >::type & ←  
operator=(  
    basic_seq_packet_socket< Protocol, SeqPacketSocketService1 > && other);
```

This assignment operator moves a sequenced packet socket from one object to another.

Parameters

other The other `basic_seq_packet_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_seq_packet_socket(io_service&)` constructor.

5.72.46 basic_seq_packet_socket::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.47 basic_seq_packet_socket::receive

Receive some data on the socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags & out_flags);

template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags in_flags,
    socket_base::message_flags & out_flags);
```

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags in_flags,
    socket_base::message_flags & out_flags,
    asio::error_code & ec);
```

5.72.47.1 basic_seq_packet_socket::receive (1 of 3 overloads)

Receive some data on the socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags & out_flags);
```

This function is used to receive data on the sequenced packet socket. The function call will block until data has been received successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

out_flags After the receive call completes, contains flags associated with the received data. For example, if the `socket_base::message_end_of_record` bit is set then the received data marks the end of a record.

Return Value

The number of bytes received.

Exceptions

`asio::system_error` Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Example

To receive into a single data buffer use the `buffer` function as follows:

```
socket.receive(asio::buffer(data, size), out_flags);
```

See the `buffer` documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.72.47.2 basic_seq_packet_socket::receive (2 of 3 overloads)

Receive some data on the socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags in_flags,
    socket_base::message_flags & out_flags);
```

This function is used to receive data on the sequenced packet socket. The function call will block until data has been received successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

in_flags Flags specifying how the receive call is to be made.

out_flags After the receive call completes, contains flags associated with the received data. For example, if the `socket_base::message_end_of_record` bit is set then the received data marks the end of a record.

Return Value

The number of bytes received.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The receive operation may not receive all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

Example

To receive into a single data buffer use the `buffer` function as follows:

```
socket.receive(asio::buffer(data, size), 0, out_flags);
```

See the `buffer` documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.72.47.3 basic_seq_packet_socket::receive (3 of 3 overloads)

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags in_flags,
    socket_base::message_flags & out_flags,
    asio::error_code & ec);
```

This function is used to receive data on the sequenced packet socket. The function call will block until data has been received successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

in_flags Flags specifying how the receive call is to be made.

out_flags After the receive call completes, contains flags associated with the received data. For example, if the `socket_base::message_end_of_record` bit is set then the received data marks the end of a record.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes received. Returns 0 if an error occurred.

Remarks

The receive operation may not receive all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

5.72.48 basic_seq_packet_socket::receive_buffer_size

Inherited from socket_base.

Socket option for the receive buffer size of a socket.

```
typedef implementation_defined receive_buffer_size;
```

Implements the SOL_SOCKET/SO_RCVBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.49 basic_seq_packet_socket::receive_low_watermark

Inherited from socket_base.

Socket option for the receive low watermark.

```
typedef implementation_defined receive_low_watermark;
```

Implements the SOL_SOCKET/SO_RCVLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.50 basic_seq_packet_socket::remote_endpoint

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;  
  
endpoint_type remote_endpoint(  
    asio::error_code & ec) const;
```

5.72.50.1 basic_seq_packet_socket::remote_endpoint (1 of 2 overloads)

Inherited from basic_socket.

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;
```

This function is used to obtain the remote endpoint of the socket.

Return Value

An object that represents the remote endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);  
...  
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint();
```

5.72.50.2 basic_seq_packet_socket::remote_endpoint (2 of 2 overloads)

Inherited from basic_socket.

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint(  
    asio::error_code & ec) const;
```

This function is used to obtain the remote endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the remote endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.72.51 basic_seq_packet_socket::reuse_address

Inherited from `socket_base`.

Socket option to allow the socket to be bound to an address that is already in use.

```
typedef implementation_defined reuse_address;
```

Implements the SOL_SOCKET/SO_REUSEADDR socket option.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: `asio/basic_seq_packet_socket.hpp`

Convenience header: `asio.hpp`

5.72.52 basic_seq_packet_socket::send

Send some data on the socket.

```

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags);

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);

```

5.72.52.1 basic_seq_packet_socket::send (1 of 2 overloads)

Send some data on the socket.

```

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags);

```

This function is used to send data on the sequenced packet socket. The function call will block until the data has been sent successfully, or an until error occurs.

Parameters

buffers One or more data buffers to be sent on the socket.

flags Flags specifying how the send call is to be made.

Return Value

The number of bytes sent.

Exceptions

asio::system_error Thrown on failure.

Example

To send a single data buffer use the **buffer** function as follows:

```
socket.send(asio::buffer(data, size), 0);
```

See the **buffer** documentation for information on sending multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

5.72.52.2 basic_seq_packet_socket::send (2 of 2 overloads)

Send some data on the socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to send data on the sequenced packet socket. The function call will block until the data has been sent successfully, or an error occurs.

Parameters

buffers One or more data buffers to be sent on the socket.

flags Flags specifying how the send call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes sent. Returns 0 if an error occurred.

Remarks

The send operation may not transmit all of the data to the peer. Consider using the [write](#) function if you need to ensure that all data is written before the blocking operation completes.

5.72.53 basic_seq_packet_socket::send_buffer_size

Inherited from socket_base.

Socket option for the send buffer size of a socket.

```
typedef implementation_defined send_buffer_size;
```

Implements the SOL_SOCKET/SO_SNDBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.54 basic_seq_packet_socket::send_low_watermark

Inherited from socket_base.

Socket option for the send low watermark.

```
typedef implementation_defined send_low_watermark;
```

Implements the SOL_SOCKET/SO SNDLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.55 basic_seq_packet_socket::service

Inherited from basic_io_object.

(Deprecated: Use get_service().) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.72.56 basic_seq_packet_socket::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef SeqPacketSocketService service_type;
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.57 basic_seq_packet_socket::set_option

Set an option on the socket.

```
void set_option(
    const SettableSocketOption & option);

asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.72.57.1 basic_seq_packet_socket::set_option (1 of 2 overloads)

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.72.57.2 basic_seq_packet_socket::set_option (2 of 2 overloads)

Inherited from `basic_socket`.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

ec Set to indicate what error occurred, if any.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.72.58 basic_seq_packet_socket::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.72.58.1 basic_seq_packet_socket::shutdown (1 of 2 overloads)

Inherited from `basic_socket`.

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

Exceptions

asio::system_error Thrown on failure.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
socket.shutdown(asio::ip::tcp::socket::shutdown_send);
```

5.72.58.2 basic_seq_packet_socket::shutdown (2 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

ec Set to indicate what error occurred, if any.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);
if (ec)
{
    // An error occurred.
}
```

5.72.59 basic_seq_packet_socket::shutdown_type

Inherited from socket_base.

Different ways a socket may be shutdown.

```
enum shutdown_type
```

Values

shutdown_receive Shutdown the receive side of the socket.

shutdown_send Shutdown the send side of the socket.

shutdown_both Shutdown both send and receive on the socket.

5.73 basic_serial_port

Provides serial port functionality.

```
template<
    typename SerialPortService = serial_port_service>
class basic_serial_port :
    public basic_io_object< SerialPortService >,
    public serial_port_base
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_serial_port is always the lowest layer.
native_handle_type	The native representation of a serial port.
native_type	(Deprecated: Use native_handle_type.) The native representation of a serial port.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native serial port to the serial port.
async_read_some	Start an asynchronous read.
async_write_some	Start an asynchronous write.
basic_serial_port	Construct a basic_serial_port without opening it. Construct and open a basic_serial_port. Construct a basic_serial_port on an existing native serial port. Move-construct a basic_serial_port from another.
cancel	Cancel all asynchronous operations associated with the serial port.

Name	Description
close	Close the serial port.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the serial port.
is_open	Determine whether the serial port is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native serial port representation.
native_handle	Get the native serial port representation.
open	Open the serial port using the specified device name.
operator=	Move-assign a basic_serial_port from another.
read_some	Read some data from the serial port.
send_break	Send a break sequence to the serial port.
set_option	Set an option on the serial port.
write_some	Write some data to the serial port.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_serial_port` class template provides functionality that is common to all serial ports.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/basic_serial_port.hpp

Convenience header: asio.hpp

5.73.1 basic_serial_port::assign

Assign an existing native serial port to the serial port.

```
void assign(
    const native_handle_type & native_serial_port);

asio::error_code assign(
    const native_handle_type & native_serial_port,
    asio::error_code & ec);
```

5.73.1.1 basic_serial_port::assign (1 of 2 overloads)

Assign an existing native serial port to the serial port.

```
void assign(
    const native_handle_type & native_serial_port);
```

5.73.1.2 basic_serial_port::assign (2 of 2 overloads)

Assign an existing native serial port to the serial port.

```
asio::error_code assign(
    const native_handle_type & native_serial_port,
    asio::error_code & ec);
```

5.73.2 basic_serial_port::async_read_some

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void or deduced async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

This function is used to asynchronously read data from the serial port. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be read. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(  
    const asio::error_code& error, // Result of operation.  
    std::size_t bytes_transferred           // Number of bytes read.  
) ;
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The read operation may not read all of the requested number of bytes. Consider using the `async_read` function if you need to ensure that the requested amount of data is read before the asynchronous operation completes.

Example

To read into a single data buffer use the `buffer` function as follows:

```
serial_port.async_read_some(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.73.3 basic_serial_port::async_write_some

Start an asynchronous write.

```
template<  
    typename ConstBufferSequence,  
    typename WriteHandler>  
void-or-deduced async_write_some(  
    const ConstBufferSequence & buffers,  
    WriteHandler handler);
```

This function is used to asynchronously write data to the serial port. The function call always returns immediately.

Parameters

buffers One or more data buffers to be written to the serial port. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(  
    const asio::error_code& error, // Result of operation.  
    std::size_t bytes_transferred           // Number of bytes written.  
) ;
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The write operation may not transmit all of the data to the peer. Consider using the `async_write` function if you need to ensure that all data is written before the asynchronous operation completes.

Example

To write a single data buffer use the `buffer` function as follows:

```
serial_port.async_write_some(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.73.4 basic_serial_port::basic_serial_port

Construct a `basic_serial_port` without opening it.

```
explicit basic_serial_port(
    asio::io_service & io_service);
```

Construct and open a `basic_serial_port`.

```
explicit basic_serial_port(
    asio::io_service & io_service,
    const char * device);

explicit basic_serial_port(
    asio::io_service & io_service,
    const std::string & device);
```

Construct a `basic_serial_port` on an existing native serial port.

```
basic_serial_port(
    asio::io_service & io_service,
    const native_handle_type & native_serial_port);
```

Move-construct a `basic_serial_port` from another.

```
basic_serial_port(
    basic_serial_port && other);
```

5.73.4.1 basic_serial_port::basic_serial_port (1 of 5 overloads)

Construct a `basic_serial_port` without opening it.

```
basic_serial_port(
    asio::io_service & io_service);
```

This constructor creates a serial port without opening it.

Parameters

io_service The `io_service` object that the serial port will use to dispatch handlers for any asynchronous operations performed on the port.

5.73.4.2 `basic_serial_port::basic_serial_port (2 of 5 overloads)`

Construct and open a `basic_serial_port`.

```
basic_serial_port(
    asio::io_service & io_service,
    const char * device);
```

This constructor creates and opens a serial port for the specified device name.

Parameters

io_service The `io_service` object that the serial port will use to dispatch handlers for any asynchronous operations performed on the port.

device The platform-specific device name for this serial port.

5.73.4.3 `basic_serial_port::basic_serial_port (3 of 5 overloads)`

Construct and open a `basic_serial_port`.

```
basic_serial_port(
    asio::io_service & io_service,
    const std::string & device);
```

This constructor creates and opens a serial port for the specified device name.

Parameters

io_service The `io_service` object that the serial port will use to dispatch handlers for any asynchronous operations performed on the port.

device The platform-specific device name for this serial port.

5.73.4.4 `basic_serial_port::basic_serial_port (4 of 5 overloads)`

Construct a `basic_serial_port` on an existing native serial port.

```
basic_serial_port(
    asio::io_service & io_service,
    const native_handle_type & native_serial_port);
```

This constructor creates a serial port object to hold an existing native serial port.

Parameters

io_service The `io_service` object that the serial port will use to dispatch handlers for any asynchronous operations performed on the port.

native_serial_port A native serial port.

Exceptions

`asio::system_error` Thrown on failure.

5.73.4.5 `basic_serial_port::basic_serial_port` (5 of 5 overloads)

Move-construct a `basic_serial_port` from another.

```
basic_serial_port(
    basic_serial_port && other);
```

This constructor moves a serial port from one object to another.

Parameters

`other` The other `basic_serial_port` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_serial_port(io_service&)` constructor.

5.73.5 `basic_serial_port::cancel`

Cancel all asynchronous operations associated with the serial port.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.73.5.1 `basic_serial_port::cancel` (1 of 2 overloads)

Cancel all asynchronous operations associated with the serial port.

```
void cancel();
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.73.5.2 `basic_serial_port::cancel` (2 of 2 overloads)

Cancel all asynchronous operations associated with the serial port.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

5.73.6 basic_serial_port::close

Close the serial port.

```
void close();  
  
asio::error_code close(  
    asio::error_code & ec);
```

5.73.6.1 basic_serial_port::close (1 of 2 overloads)

Close the serial port.

```
void close();
```

This function is used to close the serial port. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.73.6.2 basic_serial_port::close (2 of 2 overloads)

Close the serial port.

```
asio::error_code close(  
    asio::error_code & ec);
```

This function is used to close the serial port. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

5.73.7 basic_serial_port::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.73.7.1 basic_serial_port::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.73.7.2 basic_serial_port::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.73.8 basic_serial_port::get_io_service

Inherited from basic_io_object.

Get the [io_service](#) associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the [io_service](#) object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the [io_service](#) object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.73.9 basic_serial_port::get_option

Get an option from the serial port.

```
template<
    typename GettableSerialPortOption>
void get_option(
    GettableSerialPortOption & option);

template<
    typename GettableSerialPortOption>
asio::error_code get_option(
    GettableSerialPortOption & option,
    asio::error_code & ec);
```

5.73.9.1 basic_serial_port::get_option (1 of 2 overloads)

Get an option from the serial port.

```
template<
    typename GettableSerialPortOption>
void get_option(
    GettableSerialPortOption & option);
```

This function is used to get the current value of an option on the serial port.

Parameters

option The option value to be obtained from the serial port.

Exceptions

asio::system_error Thrown on failure.

5.73.9.2 basic_serial_port::get_option (2 of 2 overloads)

Get an option from the serial port.

```
template<
    typename GettableSerialPortOption>
asio::error_code get_option(
    GettableSerialPortOption & option,
    asio::error_code & ec);
```

This function is used to get the current value of an option on the serial port.

Parameters

option The option value to be obtained from the serial port.

ec Set to indicate what error occurred, if any.

5.73.10 basic_serial_port::get_service

Get the service associated with the I/O object.

```
service_type & get_service();  
  
const service_type & get_service() const;
```

5.73.10.1 basic_serial_port::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.73.10.2 basic_serial_port::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.73.11 basic_serial_port::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.73.12 basic_serial_port::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_serial_port.hpp

Convenience header: asio.hpp

5.73.13 basic_serial_port::is_open

Determine whether the serial port is open.

```
bool is_open() const;
```

5.73.14 basic_serial_port::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.73.14.1 basic_serial_port::lowest_layer (1 of 2 overloads)

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a `basic_serial_port` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.73.14.2 `basic_serial_port::lowest_layer` (2 of 2 overloads)

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a `basic_serial_port` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.73.15 `basic_serial_port::lowest_layer_type`

A `basic_serial_port` is always the lowest layer.

```
typedef basic_serial_port< SerialPortService > lowest_layer_type;
```

Types

Name	Description
<code>implementation_type</code>	The underlying implementation type of I/O object.
<code>lowest_layer_type</code>	A <code>basic_serial_port</code> is always the lowest layer.
<code>native_handle_type</code>	The native representation of a serial port.
<code>native_type</code>	(Deprecated: Use <code>native_handle_type</code> .) The native representation of a serial port.
<code>service_type</code>	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
<code>assign</code>	Assign an existing native serial port to the serial port.
<code>async_read_some</code>	Start an asynchronous read.
<code>async_write_some</code>	Start an asynchronous write.
<code>basic_serial_port</code>	Construct a <code>basic_serial_port</code> without opening it. Construct and open a <code>basic_serial_port</code> . Construct a <code>basic_serial_port</code> on an existing native serial port. Move-construct a <code>basic_serial_port</code> from another.
<code>cancel</code>	Cancel all asynchronous operations associated with the serial port.

Name	Description
close	Close the serial port.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the serial port.
is_open	Determine whether the serial port is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native serial port representation.
native_handle	Get the native serial port representation.
open	Open the serial port using the specified device name.
operator=	Move-assign a basic_serial_port from another.
read_some	Read some data from the serial port.
send_break	Send a break sequence to the serial port.
set_option	Set an option on the serial port.
write_some	Write some data to the serial port.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_serial_port` class template provides functionality that is common to all serial ports.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/basic_serial_port.hpp

Convenience header: asio.hpp

5.73.16 basic_serial_port::native

(Deprecated: Use native_handle().) Get the native serial port representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the serial port. This is intended to allow access to native serial port functionality that is not otherwise provided.

5.73.17 basic_serial_port::native_handle

Get the native serial port representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the serial port. This is intended to allow access to native serial port functionality that is not otherwise provided.

5.73.18 basic_serial_port::native_handle_type

The native representation of a serial port.

```
typedef SerialPortService::native_handle_type native_handle_type;
```

Requirements

Header: asio/basic_serial_port.hpp

Convenience header: asio.hpp

5.73.19 basic_serial_port::native_type

(Deprecated: Use native_handle_type.) The native representation of a serial port.

```
typedef SerialPortService::native_handle_type native_type;
```

Requirements

Header: asio/basic_serial_port.hpp

Convenience header: asio.hpp

5.73.20 basic_serial_port::open

Open the serial port using the specified device name.

```
void open(
    const std::string & device);

asio::error_code open(
    const std::string & device,
    asio::error_code & ec);
```

5.73.20.1 basic_serial_port::open (1 of 2 overloads)

Open the serial port using the specified device name.

```
void open(
    const std::string & device);
```

This function opens the serial port for the specified device name.

Parameters

device The platform-specific device name.

Exceptions

asio::system_error Thrown on failure.

5.73.20.2 basic_serial_port::open (2 of 2 overloads)

Open the serial port using the specified device name.

```
asio::error_code open(
    const std::string & device,
    asio::error_code & ec);
```

This function opens the serial port using the given platform-specific device name.

Parameters

device The platform-specific device name.

ec Set the indicate what error occurred, if any.

5.73.21 basic_serial_port::operator=

Move-assign a **basic_serial_port** from another.

```
basic_serial_port & operator=(
    basic_serial_port && other);
```

This assignment operator moves a serial port from one object to another.

Parameters

other The other `basic_serial_port` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_serial_port(io_service&)` constructor.

5.73.22 `basic_serial_port::read_some`

Read some data from the serial port.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);

template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.73.22.1 `basic_serial_port::read_some (1 of 2 overloads)`

Read some data from the serial port.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);
```

This function is used to read data from the serial port. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be read.

Return Value

The number of bytes read.

Exceptions

`asio::system_error` Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The `read_some` operation may not read all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

Example

To read into a single data buffer use the `buffer` function as follows:

```
serial_port.read_some(asio::buffer(data, size));
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.73.22.2 basic_serial_port::read_some (2 of 2 overloads)

Read some data from the serial port.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to read data from the serial port. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be read.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes read. Returns 0 if an error occurred.

Remarks

The `read_some` operation may not read all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

5.73.23 basic_serial_port::send_break

Send a break sequence to the serial port.

```
void send_break();

asio::error_code send_break(
    asio::error_code & ec);
```

5.73.23.1 basic_serial_port::send_break (1 of 2 overloads)

Send a break sequence to the serial port.

```
void send_break();
```

This function causes a break sequence of platform-specific duration to be sent out the serial port.

Exceptions

asio::system_error Thrown on failure.

5.73.23.2 basic_serial_port::send_break (2 of 2 overloads)

Send a break sequence to the serial port.

```
asio::error_code send_break(
    asio::error_code & ec);
```

This function causes a break sequence of platform-specific duration to be sent out the serial port.

Parameters

ec Set to indicate what error occurred, if any.

5.73.24 basic_serial_port::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.73.25 basic_serial_port::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef SerialPortService service_type;
```

Requirements

Header: asio/basic_serial_port.hpp

Convenience header: asio.hpp

5.73.26 basic_serial_port::set_option

Set an option on the serial port.

```
template<
    typename SettableSerialPortOption>
void set_option(
    const SettableSerialPortOption & option);

template<
    typename SettableSerialPortOption>
asio::error_code set_option(
    const SettableSerialPortOption & option,
    asio::error_code & ec);
```

5.73.26.1 basic_serial_port::set_option (1 of 2 overloads)

Set an option on the serial port.

```
template<
    typename SettableSerialPortOption>
void set_option(
    const SettableSerialPortOption & option);
```

This function is used to set an option on the serial port.

Parameters

option The option value to be set on the serial port.

Exceptions

asio::system_error Thrown on failure.

5.73.26.2 basic_serial_port::set_option (2 of 2 overloads)

Set an option on the serial port.

```
template<
    typename SettableSerialPortOption>
asio::error_code set_option(
    const SettableSerialPortOption & option,
    asio::error_code & ec);
```

This function is used to set an option on the serial port.

Parameters

option The option value to be set on the serial port.

ec Set to indicate what error occurred, if any.

5.73.27 basic_serial_port::write_some

Write some data to the serial port.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.73.27.1 basic_serial_port::write_some (1 of 2 overloads)

Write some data to the serial port.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

This function is used to write data to the serial port. The function call will block until one or more bytes of the data has been written successfully, or until an error occurs.

Parameters

buffers One or more data buffers to be written to the serial port.

Return Value

The number of bytes written.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The `write_some` operation may not transmit all of the data to the peer. Consider using the `write` function if you need to ensure that all data is written before the blocking operation completes.

Example

To write a single data buffer use the `buffer` function as follows:

```
serial_port.write_some(asio::buffer(data, size));
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.73.27.2 basic_serial_port::write_some (2 of 2 overloads)

Write some data to the serial port.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to write data to the serial port. The function call will block until one or more bytes of the data has been written successfully, or until an error occurs.

Parameters

buffers One or more data buffers to be written to the serial port.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes written. Returns 0 if an error occurred.

Remarks

The write_some operation may not transmit all of the data to the peer. Consider using the [write](#) function if you need to ensure that all data is written before the blocking operation completes.

5.74 basic_signal_set

Provides signal functionality.

```
template<
    typename SignalSetService = signal_set_service>
class basic_signal_set :
    public basic_io_object< SignalSetService >
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
add	Add a signal to a signal_set.

Name	Description
async_wait	Start an asynchronous operation to wait for a signal to be delivered.
basic_signal_set	Construct a signal set without adding any signals. Construct a signal set and add one signal. Construct a signal set and add two signals. Construct a signal set and add three signals.
cancel	Cancel all operations associated with the signal set.
clear	Remove all signals from a signal_set.
get_io_service	Get the io_service associated with the object.
remove	Remove a signal from a signal_set.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_signal_set` class template provides the ability to perform an asynchronous wait for one or more signals to occur.

Most applications will use the `signal_set` typedef.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Example

Performing an asynchronous wait:

```

void handler(
    const asio::error_code& error,
    int signal_number)
{
    if (!error)
    {
        // A signal occurred.
    }
}

...

// Construct a signal set registered for process termination.
asio::signal_set signals(io_service, SIGINT, SIGTERM);

// Start an asynchronous wait for one of the signals to occur.
signals.async_wait(handler);

```

Queueing of signal notifications

If a signal is registered with a signal_set, and the signal occurs when there are no waiting handlers, then the signal notification is queued. The next async_wait operation on that signal_set will dequeue the notification. If multiple notifications are queued, subsequent async_wait operations dequeue them one at a time. Signal notifications are dequeued in order of ascending signal number.

If a signal number is removed from a signal_set (using the `remove` or `erase` member functions) then any queued notifications for that signal are discarded.

Multiple registration of signals

The same signal number may be registered with different signal_set objects. When the signal occurs, one handler is called for each signal_set object.

Note that multiple registration only works for signals that are registered using Asio. The application must not also register a signal handler using functions such as `signal()` or `sigaction()`.

Signal masking on POSIX platforms

POSIX allows signals to be blocked using functions such as `sigprocmask()` and `pthread_sigmask()`. For signals to be delivered, programs must ensure that any signals registered using signal_set objects are unblocked in at least one thread.

Requirements

Header: `asio/basic_signal_set.hpp`

Convenience header: `asio.hpp`

5.74.1 basic_signal_set::add

Add a signal to a signal_set.

```

void add(
    int signal_number);

asio::error_code add(
    int signal_number,
    asio::error_code & ec);

```

5.74.1.1 basic_signal_set::add (1 of 2 overloads)

Add a signal to a signal_set.

```
void add(  
    int signal_number);
```

This function adds the specified signal to the set. It has no effect if the signal is already in the set.

Parameters

signal_number The signal to be added to the set.

Exceptions

asio::system_error Thrown on failure.

5.74.1.2 basic_signal_set::add (2 of 2 overloads)

Add a signal to a signal_set.

```
asio::error_code add(  
    int signal_number,  
    asio::error_code & ec);
```

This function adds the specified signal to the set. It has no effect if the signal is already in the set.

Parameters

signal_number The signal to be added to the set.

ec Set to indicate what error occurred, if any.

5.74.2 basic_signal_set::async_wait

Start an asynchronous operation to wait for a signal to be delivered.

```
template<  
    typename SignalHandler>  
void or deduced async_wait(  
    SignalHandler handler);
```

This function may be used to initiate an asynchronous wait against the signal set. It always returns immediately.

For each call to `async_wait()`, the supplied handler will be called exactly once. The handler will be called when:

- One of the registered signals in the signal set occurs; or
- The signal set was cancelled, in which case the handler is passed the error code `asio::error::operation_aborted`.

Parameters

handler The handler to be called when the signal occurs. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(  
    const asio::error_code& error, // Result of operation.  
    int signal_number // Indicates which signal occurred.  
) ;
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.74.3 basic_signal_set::basic_signal_set

Construct a signal set without adding any signals.

```
explicit basic_signal_set(  
    asio::io_service & io_service);
```

Construct a signal set and add one signal.

```
basic_signal_set(  
    asio::io_service & io_service,  
    int signal_number_1);
```

Construct a signal set and add two signals.

```
basic_signal_set(  
    asio::io_service & io_service,  
    int signal_number_1,  
    int signal_number_2);
```

Construct a signal set and add three signals.

```
basic_signal_set(  
    asio::io_service & io_service,  
    int signal_number_1,  
    int signal_number_2,  
    int signal_number_3);
```

5.74.3.1 basic_signal_set::basic_signal_set (1 of 4 overloads)

Construct a signal set without adding any signals.

```
basic_signal_set(  
    asio::io_service & io_service);
```

This constructor creates a signal set without registering for any signals.

Parameters

io_service The `io_service` object that the signal set will use to dispatch handlers for any asynchronous operations performed on the set.

5.74.3.2 basic_signal_set::basic_signal_set (2 of 4 overloads)

Construct a signal set and add one signal.

```
basic_signal_set(
    asio::io_service & io_service,
    int signal_number_1);
```

This constructor creates a signal set and registers for one signal.

Parameters

io_service The `io_service` object that the signal set will use to dispatch handlers for any asynchronous operations performed on the set.

signal_number_1 The signal number to be added.

Remarks

This constructor is equivalent to performing:

```
asio::signal_set signals(io_service);
signals.add(signal_number_1);
```

5.74.3.3 basic_signal_set::basic_signal_set (3 of 4 overloads)

Construct a signal set and add two signals.

```
basic_signal_set(
    asio::io_service & io_service,
    int signal_number_1,
    int signal_number_2);
```

This constructor creates a signal set and registers for two signals.

Parameters

io_service The `io_service` object that the signal set will use to dispatch handlers for any asynchronous operations performed on the set.

signal_number_1 The first signal number to be added.

signal_number_2 The second signal number to be added.

Remarks

This constructor is equivalent to performing:

```
asio::signal_set signals(io_service);
signals.add(signal_number_1);
signals.add(signal_number_2);
```

5.74.3.4 basic_signal_set::basic_signal_set (4 of 4 overloads)

Construct a signal set and add three signals.

```
basic_signal_set(
    asio::io_service & io_service,
    int signal_number_1,
    int signal_number_2,
    int signal_number_3);
```

This constructor creates a signal set and registers for three signals.

Parameters

io_service The `io_service` object that the signal set will use to dispatch handlers for any asynchronous operations performed on the set.

signal_number_1 The first signal number to be added.

signal_number_2 The second signal number to be added.

signal_number_3 The third signal number to be added.

Remarks

This constructor is equivalent to performing:

```
asio::signal_set signals(io_service);
signals.add(signal_number_1);
signals.add(signal_number_2);
signals.add(signal_number_3);
```

5.74.4 basic_signal_set::cancel

Cancel all operations associated with the signal set.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.74.4.1 basic_signal_set::cancel (1 of 2 overloads)

Cancel all operations associated with the signal set.

```
void cancel();
```

This function forces the completion of any pending asynchronous wait operations against the signal set. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Cancellation does not alter the set of registered signals.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

If a registered signal occurred before `cancel()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.74.4.2 `basic_signal_set::cancel (2 of 2 overloads)`

Cancel all operations associated with the signal set.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function forces the completion of any pending asynchronous wait operations against the signal set. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Cancellation does not alter the set of registered signals.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

If a registered signal occurred before `cancel()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.74.5 `basic_signal_set::clear`

Remove all signals from a `signal_set`.

```
void clear();

asio::error_code clear(
    asio::error_code & ec);
```

5.74.5.1 `basic_signal_set::clear (1 of 2 overloads)`

Remove all signals from a `signal_set`.

```
void clear();
```

This function removes all signals from the set. It has no effect if the set is already empty.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Removes all queued notifications.

5.74.5.2 `basic_signal_set::clear (2 of 2 overloads)`

Remove all signals from a signal_set.

```
asio::error_code clear(  
    asio::error_code & ec);
```

This function removes all signals from the set. It has no effect if the set is already empty.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

Removes all queued notifications.

5.74.6 `basic_signal_set::get_implementation`

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.74.6.1 `basic_signal_set::get_implementation (1 of 2 overloads)`

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.74.6.2 `basic_signal_set::get_implementation (2 of 2 overloads)`

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.74.7 basic_signal_set::get_io_service

Inherited from basic_io_object.

Get the `io_service` associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the `io_service` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_service` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.74.8 basic_signal_set::get_service

Get the service associated with the I/O object.

```
service_type & get_service();

const service_type & get_service() const;
```

5.74.8.1 basic_signal_set::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.74.8.2 basic_signal_set::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.74.9 basic_signal_set::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.74.10 basic_signal_set::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_signal_set.hpp

Convenience header: asio.hpp

5.74.11 basic_signal_set::remove

Remove a signal from a signal_set.

```
void remove(
    int signal_number);

asio::error_code remove(
    int signal_number,
    asio::error_code & ec);
```

5.74.11.1 basic_signal_set::remove (1 of 2 overloads)

Remove a signal from a signal_set.

```
void remove(
    int signal_number);
```

This function removes the specified signal from the set. It has no effect if the signal is not in the set.

Parameters

signal_number The signal to be removed from the set.

Exceptions

asio::system_error Thrown on failure.

Remarks

Removes any notifications that have been queued for the specified signal number.

5.74.11.2 basic_signal_set::remove (2 of 2 overloads)

Remove a signal from a signal_set.

```
asio::error_code remove(
    int signal_number,
    asio::error_code & ec);
```

This function removes the specified signal from the set. It has no effect if the signal is not in the set.

Parameters

signal_number The signal to be removed from the set.

ec Set to indicate what error occurred, if any.

Remarks

Removes any notifications that have been queued for the specified signal number.

5.74.12 basic_signal_set::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.74.13 basic_signal_set::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef SignalSetService service_type;
```

Requirements

Header: asio/basic_signal_set.hpp

Convenience header: asio.hpp

5.75 basic_socket

Provides socket functionality.

```
template<
    typename Protocol,
    typename SocketService>
class basic_socket :
    public basic_io_object< SocketService >,
    public socket_base
```

Types

Name	Description
<code>broadcast</code>	Socket option to permit sending of broadcast messages.
<code>bytes_readable</code>	IO control command to get the amount of data that can be read without blocking.
<code>debug</code>	Socket option to enable socket-level debugging.
<code>do_not_route</code>	Socket option to prevent routing, use local interfaces only.

Name	Description
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
at_mark	Determine whether the socket is at the out-of-band data mark.

Name	Description
available	Determine the number of bytes available for reading.
basic_socket	Construct a basic_socket without opening it. Construct and open a basic_socket. Construct a basic_socket, opening it and binding it to the given local endpoint. Construct a basic_socket on an existing native socket. Move-construct a basic_socket from another. Move-construct a basic_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.
operator=	Move-assign a basic_socket from another. Move-assign a basic_socket from a socket of another protocol type.
remote_endpoint	Get the remote endpoint of the socket.

Name	Description
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
<code>~basic_socket</code>	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_socket` class template provides functionality that is common to both stream-oriented and datagram-oriented sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.1 basic_socket::assign

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);

asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.75.1.1 basic_socket::assign (1 of 2 overloads)

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

5.75.1.2 basic_socket::assign (2 of 2 overloads)

Assign an existing native socket to the socket.

```
asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.75.2 basic_socket::async_connect

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
void or deduced async_connect(
    const endpoint_type & peer_endpoint,
    ConnectHandler handler);
```

This function is used to asynchronously connect a socket to the specified remote endpoint. The function call always returns immediately.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected. Copies will be made of the endpoint object as required.

handler The handler to be called when the connection operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(  
    const asio::error_code& error // Result of operation  
) ;
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

```
void connect_handler(const asio::error_code& error)  
{  
    if (!error)  
    {  
        // Connect succeeded.  
    }  
}  
  
...  
  
asio::ip::tcp::socket socket(io_service);  
asio::ip::tcp::endpoint endpoint(  
    asio::ip::address::from_string("1.2.3.4"), 12345);  
socket.async_connect(endpoint, connect_handler);
```

5.75.3 basic_socket::at_mark

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;  
  
bool at_mark(  
    asio::error_code & ec) const;
```

5.75.3.1 basic_socket::at_mark (1 of 2 overloads)

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

Exceptions

`asio::system_error` Thrown on failure.

5.75.3.2 basic_socket::at_mark (2 of 2 overloads)

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark(  
    asio::error_code & ec) const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

5.75.4 basic_socket::available

Determine the number of bytes available for reading.

```
std::size_t available() const;  
  
std::size_t available(  
    asio::error_code & ec) const;
```

5.75.4.1 basic_socket::available (1 of 2 overloads)

Determine the number of bytes available for reading.

```
std::size_t available() const;
```

This function is used to determine the number of bytes that may be read without blocking.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

Exceptions

asio::system_error Thrown on failure.

5.75.4.2 basic_socket::available (2 of 2 overloads)

Determine the number of bytes available for reading.

```
std::size_t available(  
    asio::error_code & ec) const;
```

This function is used to determine the number of bytes that may be read without blocking.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

5.75.5 basic_socket::basic_socket

Construct a **basic_socket** without opening it.

```
explicit basic_socket(
    asio::io_service & io_service);
```

Construct and open a **basic_socket**.

```
basic_socket(
    asio::io_service & io_service,
    const protocol_type & protocol);
```

Construct a **basic_socket**, opening it and binding it to the given local endpoint.

```
basic_socket(
    asio::io_service & io_service,
    const endpoint_type & endpoint);
```

Construct a **basic_socket** on an existing native socket.

```
basic_socket(
    asio::io_service & io_service,
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

Move-construct a **basic_socket** from another.

```
basic_socket(
    basic_socket && other);
```

Move-construct a **basic_socket** from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename SocketService1>
basic_socket(
    basic_socket< Protocol1, SocketService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.75.5.1 basic_socket::basic_socket (1 of 6 overloads)

Construct a **basic_socket** without opening it.

```
basic_socket(
    asio::io_service & io_service);
```

This constructor creates a socket without opening it.

Parameters

io_service The `io_service` object that the socket will use to dispatch handlers for any asynchronous operations performed on the socket.

5.75.5.2 basic_socket::basic_socket (2 of 6 overloads)

Construct and open a `basic_socket`.

```
basic_socket(
   asio::io_service & io_service,
const protocol_type & protocol);
```

This constructor creates and opens a socket.

Parameters

io_service The `io_service` object that the socket will use to dispatch handlers for any asynchronous operations performed on the socket.

protocol An object specifying protocol parameters to be used.

Exceptions

`asio::system_error` Thrown on failure.

5.75.5.3 basic_socket::basic_socket (3 of 6 overloads)

Construct a `basic_socket`, opening it and binding it to the given local endpoint.

```
basic_socket(
   asio::io_service & io_service,
const endpoint_type & endpoint);
```

This constructor creates a socket and automatically opens it bound to the specified endpoint on the local machine. The protocol used is the protocol associated with the given endpoint.

Parameters

io_service The `io_service` object that the socket will use to dispatch handlers for any asynchronous operations performed on the socket.

endpoint An endpoint on the local machine to which the socket will be bound.

Exceptions

`asio::system_error` Thrown on failure.

5.75.5.4 basic_socket::basic_socket (4 of 6 overloads)

Construct a `basic_socket` on an existing native socket.

```
basic_socket(
   asio::io_service & io_service,
const protocol_type & protocol,
const native_handle_type & native_socket);
```

This constructor creates a socket object to hold an existing native socket.

Parameters

io_service The `io_service` object that the socket will use to dispatch handlers for any asynchronous operations performed on the socket.

protocol An object specifying protocol parameters to be used.

native_socket A native socket.

Exceptions

`asio::system_error` Thrown on failure.

5.75.5.5 `basic_socket::basic_socket (5 of 6 overloads)`

Move-construct a `basic_socket` from another.

```
basic_socket(
    basic_socket && other);
```

This constructor moves a socket from one object to another.

Parameters

other The other `basic_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_socket(io_service&)` constructor.

5.75.5.6 `basic_socket::basic_socket (6 of 6 overloads)`

Move-construct a `basic_socket` from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename SocketService1>
basic_socket(
    basic_socket< Protocol1, SocketService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

This constructor moves a socket from one object to another.

Parameters

other The other `basic_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_socket(io_service&)` constructor.

5.75.6 basic_socket::bind

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.75.6.1 basic_socket::bind (1 of 2 overloads)

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345));
```

5.75.6.2 basic_socket::bind (2 of 2 overloads)

Bind the socket to the given local endpoint.

```
asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
asio::error_code ec;
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345), ec);
if (ec)
{
    // An error occurred.
}
```

5.75.7 basic_socket::broadcast

Inherited from socket_base.

Socket option to permit sending of broadcast messages.

```
typedef implementation_defined broadcast;
```

Implements the SOL_SOCKET/SO_BROADCAST socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.8 basic_socket::bytes_readable

Inherited from socket_base.

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::bytes_readable command(true);
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.9 basic_socket::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.75.9.1 basic_socket::cancel (1 of 2 overloads)

Cancel all asynchronous operations associated with the socket.

```
void cancel();
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.75.9.2 basic_socket::cancel (2 of 2 overloads)

Cancel all asynchronous operations associated with the socket.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.75.10 basic_socket::close

Close the socket.

```
void close();

asio::error_code close(
    asio::error_code & ec);
```

5.75.10.1 basic_socket::close (1 of 2 overloads)

Close the socket.

```
void close();
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

asio::system_error Thrown on failure. Note that, even if the function indicates an error, the underlying descriptor is closed.

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.75.10.2 basic_socket::close (2 of 2 overloads)

Close the socket.

```
asio::error_code close(
    asio::error_code & ec);
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any. Note that, even if the function indicates an error, the underlying descriptor is closed.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
socket.close(ec);
if (ec)
{
    // An error occurred.
}
```

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.75.11 basic_socket::connect

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);

asio::error_code connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.75.11.1 basic_socket::connect (1 of 2 overloads)

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.connect(endpoint);
```

5.75.11.2 basic_socket::connect (2 of 2 overloads)

Connect the socket to the specified endpoint.

```
asio::error_code connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
asio::error_code ec;
socket.connect(endpoint, ec);
if (ec)
{
    // An error occurred.
}
```

5.75.12 basic_socket::debug

Inherited from socket_base.

Socket option to enable socket-level debugging.

```
typedef implementation_defined debug;
```

Implements the SOL_SOCKET/SO_DEBUG socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.13 basic_socket::do_not_route

Inherited from socket_base.

Socket option to prevent routing, use local interfaces only.

```
typedef implementation_defined do_not_route;
```

Implements the SOL_SOCKET/SO_DONTROUTE socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.14 basic_socket::enable_connection_aborted

Inherited from socket_base.

Socket option to report aborted connections on accept.

```
typedef implementation_defined enable_connection_aborted;
```

Implements a custom socket option that determines whether or not an accept operation is permitted to fail with `asio::error::connection_aborted`. By default the option is false.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.15 basic_socket::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.16 basic_socket::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.75.16.1 basic_socket::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.75.16.2 basic_socket::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.75.17 basic_socket::get_io_service

Inherited from basic_io_object.

Get the [io_service](#) associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the [io_service](#) object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the [io_service](#) object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.75.18 basic_socket::get_option

Get an option from the socket.

```
template<  
    typename GettableSocketOption>  
void get_option(  
    GettableSocketOption & option) const;  
  
template<  
    typename GettableSocketOption>  
asio::error_code get_option(  
    GettableSocketOption & option,  
    asio::error_code & ec) const;
```

5.75.18.1 basic_socket::get_option (1 of 2 overloads)

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.75.18.2 basic_socket::get_option (2 of 2 overloads)

Get an option from the socket.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
asio::error_code ec;
socket.get_option(option, ec);
if (ec)
{
    // An error occurred.
}
bool is_set = option.value();
```

5.75.19 basic_socket::get_service

Get the service associated with the I/O object.

```
service_type & get_service();

const service_type & get_service() const;
```

5.75.19.1 basic_socket::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.75.19.2 basic_socket::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.75.20 basic_socket::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.75.21 basic_socket::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.22 basic_socket::io_control

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.75.22.1 basic_socket::io_control (1 of 2 overloads)

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

5.75.22.2 basic_socket::io_control (2 of 2 overloads)

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
asio::error_code ec;
socket.io_control(command, ec);
if (ec)
{
    // An error occurred.
}
std::size_t bytes_readable = command.get();
```

5.75.23 basic_socket::is_open

Determine whether the socket is open.

```
bool is_open() const;
```

5.75.24 basic_socket::keep_alive

Inherited from socket_base.

Socket option to send keep-alives.

```
typedef implementation_defined keep_alive;
```

Implements the SOL_SOCKET/SO_KEEPALIVE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.25 basic_socket::linger

Inherited from socket_base.

Socket option to specify whether the socket lingers on close if unsent data is present.

```
typedef implementation_defined linger;
```

Implements the SOL_SOCKET/SO_LINGER socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option;
socket.get_option(option);
bool is_set = option.enabled();
unsigned short timeout = option.timeout();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.26 basic_socket::local_endpoint

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;  
  
endpoint_type local_endpoint(  
    asio::error_code & ec) const;
```

5.75.26.1 basic_socket::local_endpoint (1 of 2 overloads)

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;
```

This function is used to obtain the locally bound endpoint of the socket.

Return Value

An object that represents the local endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);  
...  
asio::ip::tcp::endpoint endpoint = socket.local_endpoint();
```

5.75.26.2 basic_socket::local_endpoint (2 of 2 overloads)

Get the local endpoint of the socket.

```
endpoint_type local_endpoint(  
    asio::error_code & ec) const;
```

This function is used to obtain the locally bound endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the local endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.local_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.75.27 basic_socket::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.75.27.1 basic_socket::lowest_layer (1 of 2 overloads)

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a **basic_socket** cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.75.27.2 basic_socket::lowest_layer (2 of 2 overloads)

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a **basic_socket** cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.75.28 basic_socket::lowest_layer_type

A **basic_socket** is always the lowest layer.

```
typedef basic_socket< Protocol, SocketService > lowest_layer_type;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_socket	Construct a basic_socket without opening it. Construct and open a basic_socket. Construct a basic_socket, opening it and binding it to the given local endpoint. Construct a basic_socket on an existing native socket. Move-construct a basic_socket from another. Move-construct a basic_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.

Name	Description
operator=	Move-assign a basic_socket from another. Move-assign a basic_socket from a socket of another protocol type.
remote_endpoint	Get the remote endpoint of the socket.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_socket	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_socket` class template provides functionality that is common to both stream-oriented and datagram-oriented sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/basic_socket.hpp`

Convenience header: `asio.hpp`

5.75.29 `basic_socket::max_connections`

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.75.30 `basic_socket::message_do_not_route`

Inherited from socket_base.

Specify that the data should not be subject to routing.

```
static const int message_do_not_route = implementation_defined;
```

5.75.31 `basic_socket::message_end_of_record`

Inherited from socket_base.

Specifies that the data marks the end of a record.

```
static const int message_end_of_record = implementation_defined;
```

5.75.32 `basic_socket::message_flags`

Inherited from socket_base.

Bitmask type for flags that can be passed to send and receive operations.

```
typedef int message_flags;
```

Requirements

Header: `asio/basic_socket.hpp`

Convenience header: `asio.hpp`

5.75.33 basic_socket::message_out_of_band

Inherited from socket_base.

Process out-of-band data.

```
static const int message_out_of_band = implementation_defined;
```

5.75.34 basic_socket::message_peek

Inherited from socket_base.

Peek at incoming data without removing it from the input queue.

```
static const int message_peek = implementation_defined;
```

5.75.35 basic_socket::native

(Deprecated: Use native_handle().) Get the native socket representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.75.36 basic_socket::native_handle

Get the native socket representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.75.37 basic_socket::native_handle_type

The native representation of a socket.

```
typedef SocketService::native_handle_type native_handle_type;
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.38 basic_socket::native_non_blocking

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

Sets the non-blocking mode of the native socket implementation.

```

void native_non_blocking(
    bool mode);

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);

```

5.75.38.1 basic_socket::native_non_blocking (1 of 3 overloads)

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

This function is used to retrieve the non-blocking mode of the underlying native socket. This mode has no effect on the behaviour of the socket object's synchronous operations.

Return Value

`true` if the underlying socket is in non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Remarks

The current non-blocking mode is cached by the socket object. Consequently, the return value may be incorrect if the non-blocking mode was set directly on the native socket.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```

template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.

```

```

errno = 0;
int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
ec = asio::error_code(n < 0 ? errno : 0,
    asio::error::get_system_category());
total_bytes_transferred_ += ec ? 0 : n;

// Retry operation immediately if interrupted by signal.
if (ec == asio::error::interrupted)
    continue;

// Check if we need to run the operation again.
if (ec == asio::error::would_block
    || ec == asio::error::try_again)
{
    // We have to wait for the socket to become ready again.
    sock_.async_write_some(asio::null_buffers(), *this);
    return;
}

if (ec || n == 0)
{
    // An error occurred, or we have reached the end of the file.
    // Either way we must exit the loop so we can call the handler.
    break;
}

// Loop around to try calling sendfile again.
}
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}

};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.75.38.2 basic_socket::native_non_blocking (2 of 3 overloads)

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);
```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If true, the underlying socket is put into non-blocking mode and direct system calls may fail with asio::error::would_block (or the equivalent system error).

Exceptions

asio::system_error Thrown on failure. If the mode is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.
                errno = 0;
                int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
                ec = asio::error_code(n < 0 ? errno : 0,
                                      asio::error::get_system_category());
                total_bytes_transferred_ += ec ? 0 : n;

                // Retry operation immediately if interrupted by signal.
                if (ec == asio::error::interrupted)
                    continue;

                // Check if we need to run the operation again.
                if (ec == asio::error::would_block
                    || ec == asio::error::try_again)
                {
                    // We have to wait for the socket to become ready again.
                    sock_.async_write_some(asio::null_buffers(), *this);
                    return;
                }

                if (ec || n == 0)
                {
                    // An error occurred, or we have reached the end of the file.
                    // Either way we must exit the loop so we can call the handler.
                    break;
                }
            }
        }
    }
};
```

```

        // Loop around to try calling sendfile again.
    }

}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}
};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.75.38.3 basic_socket::native_non_blocking (3 of 3 overloads)

Sets the non-blocking mode of the native socket implementation.

```
asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

ec Set to indicate what error occurred, if any. If the mode is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);
    }
}
```

```

if (!ec)
{
    for (;;)
    {
        // Try the system call.
        errno = 0;
        int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
        ec = asio::error_code(n < 0 ? errno : 0,
            asio::error::get_system_category());
        total_bytes_transferred_ += ec ? 0 : n;

        // Retry operation immediately if interrupted by signal.
        if (ec == asio::error::interrupted)
            continue;

        // Check if we need to run the operation again.
        if (ec == asio::error::would_block
            || ec == asio::error::try_again)
        {
            // We have to wait for the socket to become ready again.
            sock_.async_write_some(asio::null_buffers(), *this);
            return;
        }

        if (ec || n == 0)
        {
            // An error occurred, or we have reached the end of the file.
            // Either way we must exit the loop so we can call the handler.
            break;
        }

        // Loop around to try calling sendfile again.
    }
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.75.39 basic_socket::native_type

(Deprecated: Use native_handle_type.) The native representation of a socket.

```
typedef SocketService::native_handle_type native_type;
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.40 basic_socket::non_blocking

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Sets the non-blocking mode of the socket.

```
void non_blocking(
    bool mode);

asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.75.40.1 basic_socket::non_blocking (1 of 3 overloads)

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Return Value

true if the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If false, synchronous operations will block until complete.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.75.40.2 basic_socket::non_blocking (2 of 3 overloads)

Sets the non-blocking mode of the socket.

```
void non_blocking(
    bool mode);
```

Parameters

mode If true, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If false, synchronous operations will block until complete.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.75.40.3 basic_socket::non_blocking (3 of 3 overloads)

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

ec Set to indicate what error occurred, if any.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.75.41 basic_socket::non_blocking_io

Inherited from socket_base.

(Deprecated: Use `non_blocking()`) IO control command to set the blocking mode of the socket.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::non_blocking_io command(true);
socket.io_control(command);
```

Requirements

Header: `asio/basic_socket.hpp`

Convenience header: `asio.hpp`

5.75.42 basic_socket::open

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());

asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.75.42.1 basic_socket::open (1 of 2 overloads)

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying protocol parameters to be used.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
```

5.75.42.2 basic_socket::open (2 of 2 overloads)

Open the socket using the specified protocol.

```
asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying which protocol is to be used.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.75.43 basic_socket::operator=

Move-assign a **basic_socket** from another.

```
basic_socket & operator=(  
    basic_socket && other);
```

Move-assign a **basic_socket** from a socket of another protocol type.

```
template<  
    typename Protocol,  
    typename SocketService1>  
enable_if< is_convertible< Protocol, Protocol >::value, basic_socket >::type & operator=(  
    basic_socket< Protocol, SocketService1 > && other);
```

5.75.43.1 basic_socket::operator= (1 of 2 overloads)

Move-assign a **basic_socket** from another.

```
basic_socket & operator=(  
    basic_socket && other);
```

This assignment operator moves a socket from one object to another.

Parameters

other The other **basic_socket** object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_socket(io_service&)` constructor.

5.75.43.2 basic_socket::operator= (2 of 2 overloads)

Move-assign a **basic_socket** from a socket of another protocol type.

```
template<  
    typename Protocol,  
    typename SocketService1>  
enable_if< is_convertible< Protocol, Protocol >::value, basic_socket >::type & operator=(  
    basic_socket< Protocol, SocketService1 > && other);
```

This assignment operator moves a socket from one object to another.

Parameters

other The other **basic_socket** object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_socket(io_service&)` constructor.

5.75.44 basic_socket::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.45 basic_socket::receive_buffer_size

Inherited from socket_base.

Socket option for the receive buffer size of a socket.

```
typedef implementation_defined receive_buffer_size;
```

Implements the SOL_SOCKET/SO_RCVBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.46 basic_socket::receive_low_watermark

Inherited from socket_base.

Socket option for the receive low watermark.

```
typedef implementation_defined receive_low_watermark;
```

Implements the SOL_SOCKET/SO_RCVLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.47 basic_socket::remote_endpoint

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;

endpoint_type remote_endpoint(
    asio::error_code & ec) const;
```

5.75.47.1 basic_socket::remote_endpoint (1 of 2 overloads)

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;
```

This function is used to obtain the remote endpoint of the socket.

Return Value

An object that represents the remote endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint();
```

5.75.47.2 basic_socket::remote_endpoint (2 of 2 overloads)

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint(
    asio::error_code & ec) const;
```

This function is used to obtain the remote endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the remote endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.75.48 basic_socket::reuse_address

Inherited from socket_base.

Socket option to allow the socket to be bound to an address that is already in use.

```
typedef implementation_defined reuse_address;
```

Implements the SOL_SOCKET/SO_REUSEADDR socket option.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.49 basic_socket::send_buffer_size

Inherited from socket_base.

Socket option for the send buffer size of a socket.

```
typedef implementation_defined send_buffer_size;
```

Implements the SOL_SOCKET/SO_SNDBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.50 basic_socket::send_low_watermark

Inherited from socket_base.

Socket option for the send low watermark.

```
typedef implementation_defined send_low_watermark;
```

Implements the SOL_SOCKET/SO SNDLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.51 basic_socket::service

Inherited from basic_io_object.

(Deprecated: Use get_service() .) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.75.52 basic_socket::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef SocketService service_type;
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.75.53 basic_socket::set_option

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);

template<
    typename SettableSocketOption>
asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.75.53.1 basic_socket::set_option (1 of 2 overloads)

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.75.53.2 basic_socket::set_option (2 of 2 overloads)

Set an option on the socket.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

ec Set to indicate what error occurred, if any.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.75.54 basic_socket::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.75.54.1 basic_socket::shutdown (1 of 2 overloads)

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

Exceptions

asio::system_error Thrown on failure.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
socket.shutdown(asio::ip::tcp::socket::shutdown_send);
```

5.75.54.2 basic_socket::shutdown (2 of 2 overloads)

Disable sends or receives on the socket.

```
asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

ec Set to indicate what error occurred, if any.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);
if (ec)
{
    // An error occurred.
}
```

5.75.55 basic_socket::shutdown_type

Inherited from `socket_base`.

Different ways a socket may be shutdown.

```
enum shutdown_type
```

Values

shutdown_receive Shutdown the receive side of the socket.

shutdown_send Shutdown the send side of the socket.

shutdown_both Shutdown both send and receive on the socket.

5.75.56 basic_socket::~basic_socket

Protected destructor to prevent deletion through this type.

```
~basic_socket();
```

5.76 basic_socket_acceptor

Provides the ability to accept new connections.

```
template<
    typename Protocol,
    typename SocketAcceptorService = socket_acceptor_service<Protocol>>
class basic_socket_acceptor :
    public basic_io_object<SocketAcceptorService>,
    public socket_base
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.

Name	Description
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of an acceptor.
native_type	(Deprecated: Use native_handle_type.) The native representation of an acceptor.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
accept	Accept a new connection. Accept a new connection and obtain the endpoint of the peer.
assign	Assigns an existing native acceptor to the acceptor.
async_accept	Start an asynchronous accept.
basic_socket_acceptor	Construct an acceptor without opening it. Construct an open acceptor. Construct an acceptor opened on the given endpoint. Construct a basic_socket_acceptor on an existing native acceptor. Move-construct a basic_socket_acceptor from another. Move-construct a basic_socket_acceptor from an acceptor of another protocol type.
bind	Bind the acceptor to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the acceptor.
close	Close the acceptor.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the acceptor.
io_control	Perform an IO control command on the acceptor.
is_open	Determine whether the acceptor is open.
listen	Place the acceptor into the state where it will listen for new connections.
local_endpoint	Get the local endpoint of the acceptor.
native	(Deprecated: Use native_handle().) Get the native acceptor representation.
native_handle	Get the native acceptor representation.
native_non_blocking	Gets the non-blocking mode of the native acceptor implementation. Sets the non-blocking mode of the native acceptor implementation.
non_blocking	Gets the non-blocking mode of the acceptor. Sets the non-blocking mode of the acceptor.
open	Open the acceptor using the specified protocol.
operator=	Move-assign a basic_socket_acceptor from another. Move-assign a basic_socket_acceptor from an acceptor of another protocol type.

Name	Description
set_option	Set an option on the acceptor.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_socket_acceptor` class template is used for accepting new socket connections.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Example

Opening a socket acceptor with the SO_REUSEADDR option enabled:

```

asio::ip::tcp::acceptor acceptor(io_service);
asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), port);
acceptor.open(endpoint.protocol());
acceptor.set_option(asio::ip::tcp::acceptor::reuse_address(true));
acceptor.bind(endpoint);
acceptor.listen();

```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.1 basic_socket_acceptor::accept

Accept a new connection.

```

template<
    typename Protocol1,
    typename SocketService>
void accept(
    basic_socket< Protocol1, SocketService > & peer,
    typename enable_if< is_convertible< Protocol, Protocol1 >::value >::type * = 0);

template<
    typename Protocol1,
    typename SocketService>
asio::error_code accept(
    basic_socket< Protocol1, SocketService > & peer,
    asio::error_code & ec,
    typename enable_if< is_convertible< Protocol, Protocol1 >::value >::type * = 0);

```

Accept a new connection and obtain the endpoint of the peer.

```

template<
    typename SocketService>
void accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint);

template<
    typename SocketService>
asio::error_code accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint,
    asio::error_code & ec);

```

5.76.1.1 basic_socket_acceptor::accept (1 of 4 overloads)

Accept a new connection.

```

template<
    typename Protocol1,
    typename SocketService>

```

```
void accept(
    basic_socket< Protocol1, SocketService > & peer,
    typename enable_if< is_convertible< Protocol, Protocol1 >::value >::type * = 0);
```

This function is used to accept a new connection from a peer into the given socket. The function call will block until a new connection has been accepted successfully or an error occurs.

Parameters

peer The socket into which the new connection will be accepted.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::socket socket(io_service);
acceptor.accept(socket);
```

5.76.1.2 basic_socket_acceptor::accept (2 of 4 overloads)

Accept a new connection.

```
template<
    typename Protocol1,
    typename SocketService>
asio::error_code accept(
    basic_socket< Protocol1, SocketService > & peer,
    asio::error_code & ec,
    typename enable_if< is_convertible< Protocol, Protocol1 >::value >::type * = 0);
```

This function is used to accept a new connection from a peer into the given socket. The function call will block until a new connection has been accepted successfully or an error occurs.

Parameters

peer The socket into which the new connection will be accepted.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::socket socket(io_service);
asio::error_code ec;
acceptor.accept(socket, ec);
if (ec)
{
    // An error occurred.
}
```

5.76.1.3 basic_socket_acceptor::accept (3 of 4 overloads)

Accept a new connection and obtain the endpoint of the peer.

```
template<
    typename SocketService>
void accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint);
```

This function is used to accept a new connection from a peer into the given socket, and additionally provide the endpoint of the remote peer. The function call will block until a new connection has been accepted successfully or an error occurs.

Parameters

peer The socket into which the new connection will be accepted.

peer_endpoint An endpoint object which will receive the endpoint of the remote peer.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint;
acceptor.accept(socket, endpoint);
```

5.76.1.4 basic_socket_acceptor::accept (4 of 4 overloads)

Accept a new connection and obtain the endpoint of the peer.

```
template<
    typename SocketService>
asio::error_code accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

This function is used to accept a new connection from a peer into the given socket, and additionally provide the endpoint of the remote peer. The function call will block until a new connection has been accepted successfully or an error occurs.

Parameters

peer The socket into which the new connection will be accepted.

peer_endpoint An endpoint object which will receive the endpoint of the remote peer.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint;
asio::error_code ec;
acceptor.accept(socket, endpoint, ec);
if (ec)
{
    // An error occurred.
}
```

5.76.2 basic_socket_acceptor::assign

Assigns an existing native acceptor to the acceptor.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_acceptor);

asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_acceptor,
    asio::error_code & ec);
```

5.76.2.1 basic_socket_acceptor::assign (1 of 2 overloads)

Assigns an existing native acceptor to the acceptor.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_acceptor);
```

5.76.2.2 basic_socket_acceptor::assign (2 of 2 overloads)

Assigns an existing native acceptor to the acceptor.

```
asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_acceptor,
    asio::error_code & ec);
```

5.76.3 basic_socket_acceptor::async_accept

Start an asynchronous accept.

```
template<
    typename Protocol1,
    typename SocketService,
    typename AcceptHandler>
void-or-deduced async_accept(
    basic_socket< Protocol1, SocketService > & peer,
    AcceptHandler handler,
    typename enable_if< is_convertible< Protocol, Protocol1 >::value >::type * = 0);
```

```

template<
    typename SocketService,
    typename AcceptHandler>
void-or-deduced async_accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint,
    AcceptHandler handler);

```

5.76.3.1 basic_socket_acceptor::async_accept (1 of 2 overloads)

Start an asynchronous accept.

```

template<
    typename Protocol1,
    typename SocketService,
    typename AcceptHandler>
void-or-deduced async_accept(
    basic_socket< Protocol1, SocketService > & peer,
    AcceptHandler handler,
    typename enable_if< is_convertible< Protocol, Protocol1 >::value >::type * = 0);

```

This function is used to asynchronously accept a new connection into a socket. The function call always returns immediately.

Parameters

peer The socket into which the new connection will be accepted. Ownership of the peer object is retained by the caller, which must guarantee that it is valid until the handler is called.

handler The handler to be called when the accept operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    const asio::error_code& error // Result of operation.
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

```

void accept_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Accept succeeded.
    }
}

...

asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::socket socket(io_service);
acceptor.async_accept(socket, accept_handler);

```

5.76.3.2 basic_socket_acceptor::async_accept (2 of 2 overloads)

Start an asynchronous accept.

```
template<
    typename SocketService,
    typename AcceptHandler>
void-or-deduced async_accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint,
    AcceptHandler handler);
```

This function is used to asynchronously accept a new connection into a socket, and additionally obtain the endpoint of the remote peer. The function call always returns immediately.

Parameters

peer The socket into which the new connection will be accepted. Ownership of the peer object is retained by the caller, which must guarantee that it is valid until the handler is called.

peer_endpoint An endpoint object into which the endpoint of the remote peer will be written. Ownership of the peer_endpoint object is retained by the caller, which must guarantee that it is valid until the handler is called.

handler The handler to be called when the accept operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error // Result of operation.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.76.4 basic_socket_acceptor::basic_socket_acceptor

Construct an acceptor without opening it.

```
explicit basic_socket_acceptor(
    asio::io_service & io_service);
```

Construct an open acceptor.

```
basic_socket_acceptor(
    asio::io_service & io_service,
    const protocol_type & protocol);
```

Construct an acceptor opened on the given endpoint.

```
basic_socket_acceptor(
    asio::io_service & io_service,
    const endpoint_type & endpoint,
    bool reuse_addr = true);
```

Construct a `basic_socket_acceptor` on an existing native acceptor.

```
basic_socket_acceptor(
   asio::io_service & io_service,
const protocol_type & protocol,
const native_handle_type & native_acceptor);
```

Move-construct a `basic_socket_acceptor` from another.

```
basic_socket_acceptor(
    basic_socket_acceptor && other);
```

Move-construct a `basic_socket_acceptor` from an acceptor of another protocol type.

```
template<
    typename Protocol1,
    typename SocketAcceptorService1>
basic_socket_acceptor(
    basic_socket_acceptor< Protocol1, SocketAcceptorService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.76.4.1 `basic_socket_acceptor::basic_socket_acceptor` (1 of 6 overloads)

Construct an acceptor without opening it.

```
basic_socket_acceptor(
    asio::io_service & io_service);
```

This constructor creates an acceptor without opening it to listen for new connections. The `open()` function must be called before the acceptor can accept new socket connections.

Parameters

io_service The `io_service` object that the acceptor will use to dispatch handlers for any asynchronous operations performed on the acceptor.

5.76.4.2 `basic_socket_acceptor::basic_socket_acceptor` (2 of 6 overloads)

Construct an open acceptor.

```
basic_socket_acceptor(
    asio::io_service & io_service,
    const protocol_type & protocol);
```

This constructor creates an acceptor and automatically opens it.

Parameters

io_service The `io_service` object that the acceptor will use to dispatch handlers for any asynchronous operations performed on the acceptor.

protocol An object specifying protocol parameters to be used.

Exceptions

asio::system_error Thrown on failure.

5.76.4.3 basic_socket_acceptor::basic_socket_acceptor (3 of 6 overloads)

Construct an acceptor opened on the given endpoint.

```
basic_socket_acceptor(
    asio::io_service & io_service,
    const endpoint_type & endpoint,
    bool reuse_addr = true);
```

This constructor creates an acceptor and automatically opens it to listen for new connections on the specified endpoint.

Parameters

io_service The **io_service** object that the acceptor will use to dispatch handlers for any asynchronous operations performed on the acceptor.

endpoint An endpoint on the local machine on which the acceptor will listen for new connections.

reuse_addr Whether the constructor should set the socket option `socket_base::reuse_address`.

Exceptions

asio::system_error Thrown on failure.

Remarks

This constructor is equivalent to the following code:

```
basic_socket_acceptor<Protocol> acceptor(io_service);
acceptor.open(endpoint.protocol());
if (reuse_addr)
    acceptor.set_option(socket_base::reuse_address(true));
acceptor.bind(endpoint);
acceptor.listen(listen_backlog);
```

5.76.4.4 basic_socket_acceptor::basic_socket_acceptor (4 of 6 overloads)

Construct a **basic_socket_acceptor** on an existing native acceptor.

```
basic_socket_acceptor(
    asio::io_service & io_service,
    const protocol_type & protocol,
    const native_handle_type & native_acceptor);
```

This constructor creates an acceptor object to hold an existing native acceptor.

Parameters

io_service The **io_service** object that the acceptor will use to dispatch handlers for any asynchronous operations performed on the acceptor.

protocol An object specifying protocol parameters to be used.

native_acceptor A native acceptor.

Exceptions

`asio::system_error` Thrown on failure.

5.76.4.5 `basic_socket_acceptor::basic_socket_acceptor (5 of 6 overloads)`

Move-construct a `basic_socket_acceptor` from another.

```
basic_socket_acceptor(
    basic_socket_acceptor && other);
```

This constructor moves an acceptor from one object to another.

Parameters

other The other `basic_socket_acceptor` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_socket_acceptor(io_service&)` constructor.

5.76.4.6 `basic_socket_acceptor::basic_socket_acceptor (6 of 6 overloads)`

Move-construct a `basic_socket_acceptor` from an acceptor of another protocol type.

```
template<
    typename Protocol1,
    typename SocketAcceptorService1>
basic_socket_acceptor(
    basic_socket_acceptor< Protocol1, SocketAcceptorService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

This constructor moves an acceptor from one object to another.

Parameters

other The other `basic_socket_acceptor` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_socket(io_service&)` constructor.

5.76.5 `basic_socket_acceptor::bind`

Bind the acceptor to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);
```

```
asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.76.5.1 basic_socket_acceptor::bind (1 of 2 overloads)

Bind the acceptor to the given local endpoint.

```
void bind(  
    const endpoint_type & endpoint);
```

This function binds the socket acceptor to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket acceptor will be bound.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);  
asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), 12345);  
acceptor.open(endpoint.protocol());  
acceptor.bind(endpoint);
```

5.76.5.2 basic_socket_acceptor::bind (2 of 2 overloads)

Bind the acceptor to the given local endpoint.

```
asio::error_code bind(  
    const endpoint_type & endpoint,  
    asio::error_code & ec);
```

This function binds the socket acceptor to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket acceptor will be bound.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);  
asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), 12345);  
acceptor.open(endpoint.protocol());  
asio::error_code ec;  
acceptor.bind(endpoint, ec);  
if (ec)  
{  
    // An error occurred.  
}
```

5.76.6 basic_socket_acceptor::broadcast

Inherited from socket_base.

Socket option to permit sending of broadcast messages.

```
typedef implementation_defined broadcast;
```

Implements the SOL_SOCKET/SO_BROADCAST socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.7 basic_socket_acceptor::bytes_readable

Inherited from socket_base.

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::bytes_readable command(true);
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.8 basic_socket_acceptor::cancel

Cancel all asynchronous operations associated with the acceptor.

```
void cancel();  
  
asio::error_code cancel(  
    asio::error_code & ec);
```

5.76.8.1 basic_socket_acceptor::cancel (1 of 2 overloads)

Cancel all asynchronous operations associated with the acceptor.

```
void cancel();
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.76.8.2 basic_socket_acceptor::cancel (2 of 2 overloads)

Cancel all asynchronous operations associated with the acceptor.

```
asio::error_code cancel(  
    asio::error_code & ec);
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

5.76.9 basic_socket_acceptor::close

Close the acceptor.

```
void close();  
  
asio::error_code close(  
    asio::error_code & ec);
```

5.76.9.1 basic_socket_acceptor::close (1 of 2 overloads)

Close the acceptor.

```
void close();
```

This function is used to close the acceptor. Any asynchronous accept operations will be cancelled immediately.

A subsequent call to `open()` is required before the acceptor can again be used to again perform socket accept operations.

Exceptions

asio::system_error Thrown on failure.

5.76.9.2 basic_socket_acceptor::close (2 of 2 overloads)

Close the acceptor.

```
asio::error_code close(  
    asio::error_code & ec);
```

This function is used to close the acceptor. Any asynchronous accept operations will be cancelled immediately.

A subsequent call to `open()` is required before the acceptor can again be used to again perform socket accept operations.

Parameters

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);  
...  
asio::error_code ec;  
acceptor.close(ec);  
if (ec)  
{  
    // An error occurred.  
}
```

5.76.10 basic_socket_acceptor::debug

Inherited from socket_base.

Socket option to enable socket-level debugging.

```
typedef implementation_defined debug;
```

Implements the SOL_SOCKET/SO_DEBUG socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);  
...  
asio::socket_base::debug option(true);  
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);  
...  
asio::socket_base::debug option;  
socket.get_option(option);  
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.11 basic_socket_acceptor::do_not_route

Inherited from socket_base.

Socket option to prevent routing, use local interfaces only.

```
typedef implementation_defined do_not_route;
```

Implements the SOL_SOCKET/SO_DONTROUTE socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.12 basic_socket_acceptor::enable_connection_aborted

Inherited from socket_base.

Socket option to report aborted connections on accept.

```
typedef implementation_defined enable_connection_aborted;
```

Implements a custom socket option that determines whether or not an accept operation is permitted to fail with `asio::error::connection_aborted`. By default the option is false.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.13 basic_socket_acceptor::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.14 basic_socket_acceptor::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();

const implementation_type & get_implementation() const;
```

5.76.14.1 basic_socket_acceptor::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.76.14.2 basic_socket_acceptor::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.76.15 basic_socket_acceptor::get_io_service

Inherited from basic_io_object.

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the **io_service** object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the **io_service** object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.76.16 basic_socket_acceptor::get_option

Get an option from the acceptor.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option);

template<
    typename GettableSocketOption>
asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec);
```

5.76.16.1 basic_socket_acceptor::get_option (1 of 2 overloads)

Get an option from the acceptor.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option);
```

This function is used to get the current value of an option on the acceptor.

Parameters

option The option value to be obtained from the acceptor.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the value of the SOL_SOCKET/SO_REUSEADDR option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::acceptor::reuse_address option;
acceptor.get_option(option);
bool is_set = option.get();
```

5.76.16.2 basic_socket_acceptor::get_option (2 of 2 overloads)

Get an option from the acceptor.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec);
```

This function is used to get the current value of an option on the acceptor.

Parameters

option The option value to be obtained from the acceptor.

ec Set to indicate what error occurred, if any.

Example

Getting the value of the SOL_SOCKET/SO_REUSEADDR option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::acceptor::reuse_address option;
asio::error_code ec;
acceptor.get_option(option, ec);
if (ec)
{
    // An error occurred.
}
bool is_set = option.get();
```

5.76.17 basic_socket_acceptor::get_service

Get the service associated with the I/O object.

```
service_type & get_service();

const service_type & get_service() const;
```

5.76.17.1 basic_socket_acceptor::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.76.17.2 basic_socket_acceptor::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.76.18 basic_socket_acceptor::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.76.19 basic_socket_acceptor::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.20 basic_socket_acceptor::io_control

Perform an IO control command on the acceptor.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.76.20.1 basic_socket_acceptor::io_control (1 of 2 overloads)

Perform an IO control command on the acceptor.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);
```

This function is used to execute an IO control command on the acceptor.

Parameters

command The IO control command to be performed on the acceptor.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::acceptor::non_blocking_io command(true);
socket.io_control(command);
```

5.76.20.2 basic_socket_acceptor::io_control (2 of 2 overloads)

Perform an IO control command on the acceptor.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

This function is used to execute an IO control command on the acceptor.

Parameters

command The IO control command to be performed on the acceptor.

ec Set to indicate what error occurred, if any.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::acceptor::non_blocking_io command(true);
asio::error_code ec;
socket.io_control(command, ec);
if (ec)
{
    // An error occurred.
}
```

5.76.21 basic_socket_acceptor::is_open

Determine whether the acceptor is open.

```
bool is_open() const;
```

5.76.22 basic_socket_acceptor::keep_alive

Inherited from socket_base.

Socket option to send keep-alives.

```
typedef implementation_defined keep_alive;
```

Implements the SOL_SOCKET/SO_KEEPALIVE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.23 basic_socket_acceptor::linger

Inherited from socket_base.

Socket option to specify whether the socket lingers on close if unsent data is present.

```
typedef implementation_defined linger;
```

Implements the SOL_SOCKET/SO_LINGER socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option;
socket.get_option(option);
bool is_set = option.enabled();
unsigned short timeout = option.timeout();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.24 basic_socket_acceptor::listen

Place the acceptor into the state where it will listen for new connections.

```
void listen(
    int backlog = socket_base::max_connections);

asio::error_code listen(
    int backlog,
    asio::error_code & ec);
```

5.76.24.1 basic_socket_acceptor::listen (1 of 2 overloads)

Place the acceptor into the state where it will listen for new connections.

```
void listen(
    int backlog = socket_base::max_connections);
```

This function puts the socket acceptor into the state where it may accept new connections.

Parameters

backlog The maximum length of the queue of pending connections.

Exceptions

asio::system_error Thrown on failure.

5.76.24.2 basic_socket_acceptor::listen (2 of 2 overloads)

Place the acceptor into the state where it will listen for new connections.

```
asio::error_code listen(
    int backlog,
    asio::error_code & ec);
```

This function puts the socket acceptor into the state where it may accept new connections.

Parameters

backlog The maximum length of the queue of pending connections.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::error_code ec;
acceptor.listen(asio::socket_base::max_connections, ec);
if (ec)
{
    // An error occurred.
}
```

5.76.25 basic_socket_acceptor::local_endpoint

Get the local endpoint of the acceptor.

```
endpoint_type local_endpoint() const;

endpoint_type local_endpoint(
    asio::error_code & ec) const;
```

5.76.25.1 basic_socket_acceptor::local_endpoint (1 of 2 overloads)

Get the local endpoint of the acceptor.

```
endpoint_type local_endpoint() const;
```

This function is used to obtain the locally bound endpoint of the acceptor.

Return Value

An object that represents the local endpoint of the acceptor.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::endpoint endpoint = acceptor.local_endpoint();
```

5.76.25.2 basic_socket_acceptor::local_endpoint (2 of 2 overloads)

Get the local endpoint of the acceptor.

```
endpoint_type local_endpoint(
    asio::error_code & ec) const;
```

This function is used to obtain the locally bound endpoint of the acceptor.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the local endpoint of the acceptor. Returns a default-constructed endpoint object if an error occurred and the error handler did not throw an exception.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = acceptor.local_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.76.26 basic_socket_acceptor::max_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.76.27 basic_socket_acceptor::message_do_not_route

Inherited from socket_base.

Specify that the data should not be subject to routing.

```
static const int message_do_not_route = implementation_defined;
```

5.76.28 basic_socket_acceptor::message_end_of_record

Inherited from socket_base.

Specifies that the data marks the end of a record.

```
static const int message_end_of_record = implementation_defined;
```

5.76.29 basic_socket_acceptor::message_flags

Inherited from socket_base.

Bitmask type for flags that can be passed to send and receive operations.

```
typedef int message_flags;
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.30 basic_socket_acceptor::message_out_of_band

Inherited from socket_base.

Process out-of-band data.

```
static const int message_out_of_band = implementation_defined;
```

5.76.31 basic_socket_acceptor::message_peek

Inherited from socket_base.

Peek at incoming data without removing it from the input queue.

```
static const int message_peek = implementation_defined;
```

5.76.32 basic_socket_acceptor::native

(Deprecated: Use native_handle().) Get the native acceptor representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the acceptor. This is intended to allow access to native acceptor functionality that is not otherwise provided.

5.76.33 basic_socket_acceptor::native_handle

Get the native acceptor representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the acceptor. This is intended to allow access to native acceptor functionality that is not otherwise provided.

5.76.34 basic_socket_acceptor::native_handle_type

The native representation of an acceptor.

```
typedef SocketAcceptorService::native_handle_type native_handle_type;
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.35 basic_socket_acceptor::native_non_blocking

Gets the non-blocking mode of the native acceptor implementation.

```
bool native_non_blocking() const;
```

Sets the non-blocking mode of the native acceptor implementation.

```
void native_non_blocking(
    bool mode);

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.76.35.1 basic_socket_acceptor::native_non_blocking (1 of 3 overloads)

Gets the non-blocking mode of the native acceptor implementation.

```
bool native_non_blocking() const;
```

This function is used to retrieve the non-blocking mode of the underlying native acceptor. This mode has no effect on the behaviour of the acceptor object's synchronous operations.

Return Value

`true` if the underlying acceptor is in non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Remarks

The current non-blocking mode is cached by the acceptor object. Consequently, the return value may be incorrect if the non-blocking mode was set directly on the native acceptor.

5.76.35.2 basic_socket_acceptor::native_non_blocking (2 of 3 overloads)

Sets the non-blocking mode of the native acceptor implementation.

```
void native_non_blocking(
    bool mode);
```

This function is used to modify the non-blocking mode of the underlying native acceptor. It has no effect on the behaviour of the acceptor object's synchronous operations.

Parameters

mode If `true`, the underlying acceptor is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Exceptions

`asio::system_error` Thrown on failure. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

5.76.35.3 `basic_socket_acceptor::native_non_blocking` (3 of 3 overloads)

Sets the non-blocking mode of the native acceptor implementation.

```
asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

This function is used to modify the non-blocking mode of the underlying native acceptor. It has no effect on the behaviour of the acceptor object's synchronous operations.

Parameters

mode If `true`, the underlying acceptor is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

ec Set to indicate what error occurred, if any. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

5.76.36 `basic_socket_acceptor::native_type`

(Deprecated: Use `native_handle_type`.) The native representation of an acceptor.

```
typedef SocketAcceptorService::native_handle_type native_type;
```

Requirements

Header: `asio/basic_socket_acceptor.hpp`

Convenience header: `asio.hpp`

5.76.37 `basic_socket_acceptor::non_blocking`

Gets the non-blocking mode of the acceptor.

```
bool non_blocking() const;
```

Sets the non-blocking mode of the acceptor.

```
void non_blocking(
    bool mode);

asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.76.37.1 basic_socket_acceptor::non_blocking (1 of 3 overloads)

Gets the non-blocking mode of the acceptor.

```
bool non_blocking() const;
```

Return Value

true if the acceptor's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If false, synchronous operations will block until complete.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.76.37.2 basic_socket_acceptor::non_blocking (2 of 3 overloads)

Sets the non-blocking mode of the acceptor.

```
void non_blocking(  
    bool mode);
```

Parameters

mode If true, the acceptor's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If false, synchronous operations will block until complete.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.76.37.3 basic_socket_acceptor::non_blocking (3 of 3 overloads)

Sets the non-blocking mode of the acceptor.

```
asio::error_code non_blocking(  
    bool mode,  
    asio::error_code & ec);
```

Parameters

mode If true, the acceptor's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If false, synchronous operations will block until complete.

ec Set to indicate what error occurred, if any.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.76.38 basic_socket_acceptor::non_blocking_io

Inherited from socket_base.

(Deprecated: Use `non_blocking()`) IO control command to set the blocking mode of the socket.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::non_blocking_io command(true);
socket.io_control(command);
```

Requirements

Header: `asio/basic_socket_acceptor.hpp`

Convenience header: `asio.hpp`

5.76.39 basic_socket_acceptor::open

Open the acceptor using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());

asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.76.39.1 basic_socket_acceptor::open (1 of 2 overloads)

Open the acceptor using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());
```

This function opens the socket acceptor so that it will use the specified protocol.

Parameters

protocol An object specifying which protocol is to be used.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);
acceptor.open(asio::ip::tcp::v4());
```

5.76.39.2 basic_socket_acceptor::open (2 of 2 overloads)

Open the acceptor using the specified protocol.

```
asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

This function opens the socket acceptor so that it will use the specified protocol.

Parameters

protocol An object specifying which protocol is to be used.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::acceptor acceptor(io_service);
asio::error_code ec;
acceptor.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.76.40 basic_socket_acceptor::operator=

Move-assign a **basic_socket_acceptor** from another.

```
basic_socket_acceptor & operator=(
    basic_socket_acceptor && other);
```

Move-assign a **basic_socket_acceptor** from an acceptor of another protocol type.

```
template<
    typename Protocol1,
    typename SocketAcceptorService1>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_socket_acceptor >::type & ←
operator=(  
    basic_socket_acceptor< Protocol1, SocketAcceptorService1 > && other);
```

5.76.40.1 basic_socket_acceptor::operator= (1 of 2 overloads)

Move-assign a `basic_socket_acceptor` from another.

```
basic_socket_acceptor & operator=(  
    basic_socket_acceptor && other);
```

This assignment operator moves an acceptor from one object to another.

Parameters

other The other `basic_socket_acceptor` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_socket_acceptor(io_service&)` constructor.

5.76.40.2 basic_socket_acceptor::operator= (2 of 2 overloads)

Move-assign a `basic_socket_acceptor` from an acceptor of another protocol type.

```
template<  
    typename Protocol1,  
    typename SocketAcceptorService1>  
enable_if< is_convertible< Protocol1, Protocol >::value, basic_socket_acceptor >::type & ←  
operator=(  
    basic_socket_acceptor< Protocol1, SocketAcceptorService1 > && other);
```

This assignment operator moves an acceptor from one object to another.

Parameters

other The other `basic_socket_acceptor` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_socket(io_service&)` constructor.

5.76.41 basic_socket_acceptor::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.42 basic_socket_acceptor::receive_buffer_size

Inherited from socket_base.

Socket option for the receive buffer size of a socket.

```
typedef implementation_defined receive_buffer_size;
```

Implements the SOL_SOCKET/SO_RCVBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.43 basic_socket_acceptor::receive_low_watermark

Inherited from socket_base.

Socket option for the receive low watermark.

```
typedef implementation_defined receive_low_watermark;
```

Implements the SOL_SOCKET/SO_RCVLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.44 basic_socket_acceptor::reuse_address

Inherited from socket_base.

Socket option to allow the socket to be bound to an address that is already in use.

```
typedef implementation_defined reuse_address;
```

Implements the SOL_SOCKET/SO_REUSEADDR socket option.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.45 basic_socket_acceptor::send_buffer_size

Inherited from socket_base.

Socket option for the send buffer size of a socket.

```
typedef implementation_defined send_buffer_size;
```

Implements the SOL_SOCKET/SO_SNDBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.46 basic_socket_acceptor::send_low_watermark

Inherited from socket_base.

Socket option for the send low watermark.

```
typedef implementation_defined send_low_watermark;
```

Implements the SOL_SOCKET/SO_SNDLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.47 basic_socket_acceptor::service

Inherited from basic_io_object.

(Deprecated: Use get_service() .) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.76.48 basic_socket_acceptor::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef SocketAcceptorService service_type;
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.76.49 basic_socket_acceptor::set_option

Set an option on the acceptor.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);

template<
    typename SettableSocketOption>
asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.76.49.1 basic_socket_acceptor::set_option (1 of 2 overloads)

Set an option on the acceptor.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);
```

This function is used to set an option on the acceptor.

Parameters

option The new option value to be set on the acceptor.

Exceptions

asio::system_error Thrown on failure.

Example

Setting the SOL_SOCKET/SO_REUSEADDR option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::acceptor::reuse_address option(true);
acceptor.set_option(option);
```

5.76.49.2 basic_socket_acceptor::set_option (2 of 2 overloads)

Set an option on the acceptor.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

This function is used to set an option on the acceptor.

Parameters

option The new option value to be set on the acceptor.

ec Set to indicate what error occurred, if any.

Example

Setting the SOL_SOCKET/SO_REUSEADDR option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::ip::tcp::acceptor::reuse_address option(true);
asio::error_code ec;
acceptor.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.76.50 basic_socket_acceptor::shutdown_type

Inherited from socket_base.

Different ways a socket may be shutdown.

```
enum shutdown_type
```

Values

shutdown_receive Shutdown the receive side of the socket.

shutdown_send Shutdown the send side of the socket.

shutdown_both Shutdown both send and receive on the socket.

5.77 basic_socket_iostream

Iostream interface for a socket.

```
template<
    typename Protocol,
    typename StreamSocketService = stream_socket_service<Protocol>,
    typename Time = boost::posix_time::ptime,
    typename TimeTraits = asio::time_traits<Time>,
    typename TimerService = deadline_timer_service<Time, TimeTraits>>
class basic_socket_iostream
```

Types

Name	Description
duration_type	The duration type.
endpoint_type	The endpoint type.
time_type	The time type.

Member Functions

Name	Description
basic_socket_iostream	Construct a basic_socket_iostream without establishing a connection. Establish a connection to an endpoint corresponding to a resolver query.
close	Close the connection.
connect	Establish a connection to an endpoint corresponding to a resolver query.
error	Get the last error associated with the stream.
expires_at	Get the stream's expiry time as an absolute time. Set the stream's expiry time as an absolute time.
expires_from_now	Get the timer's expiry time relative to now. Set the stream's expiry time relative to now.
rdbuf	Return a pointer to the underlying streambuf.

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.77.1 basic_socket_iostream::basic_socket_iostream

Construct a `basic_socket_iostream` without establishing a connection.

```
basic_socket_iostream();
```

Establish a connection to an endpoint corresponding to a resolver query.

```
template<
    typename T1,
    ... ,
    typename TN>
explicit basic_socket_iostream(
    T1 t1,
    ... ,
    TN tn);
```

5.77.1.1 basic_socket_iostream::basic_socket_iostream (1 of 2 overloads)

Construct a `basic_socket_iostream` without establishing a connection.

```
basic_socket_iostream();
```

5.77.1.2 basic_socket_iostream::basic_socket_iostream (2 of 2 overloads)

Establish a connection to an endpoint corresponding to a resolver query.

```
template<
    typename T1,
    ... ,
    typename TN>
basic_socket_iostream(
    T1 t1,
    ... ,
    TN tn);
```

This constructor automatically establishes a connection based on the supplied resolver query parameters. The arguments are used to construct a resolver query object.

5.77.2 basic_socket_iostream::close

Close the connection.

```
void close();
```

5.77.3 basic_socket_iostream::connect

Establish a connection to an endpoint corresponding to a resolver query.

```
template<
    typename T1,
    ... ,
    typename TN>
void connect(
    T1 t1,
    ... ,
    TN tn);
```

This function automatically establishes a connection based on the supplied resolver query parameters. The arguments are used to construct a resolver query object.

5.77.4 `basic_socket_iostream::duration_type`

The duration type.

```
typedef TimeTraits::duration_type duration_type;
```

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.77.5 `basic_socket_iostream::endpoint_type`

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.77.6 `basic_socket_iostream::error`

Get the last error associated with the stream.

```
const asio::error_code & error() const;
```

Return Value

An `error_code` corresponding to the last error from the stream.

Example

To print the error associated with a failure to establish a connection:

```
tcp::iostream s("www.boost.org", "http");
if (!s)
{
    std::cout << "Error: " << s.error().message() << std::endl;
}
```

5.77.7 basic_socket_iostream::expires_at

Get the stream's expiry time as an absolute time.

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

Set the stream's expiry time as an absolute time.

```
void expires_at(
    const time_type & expiry_time);
```

5.77.7.1 basic_socket_iostream::expires_at (1 of 2 overloads)

Get the stream's expiry time as an absolute time.

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

Return Value

An absolute time value representing the stream's expiry time.

5.77.7.2 basic_socket_iostream::expires_at (2 of 2 overloads)

Set the stream's expiry time as an absolute time.

```
void expires_at(
    const time_type & expiry_time);
```

This function sets the expiry time associated with the stream. Stream operations performed after this time (where the operations cannot be completed using the internal buffers) will fail with the error `asio::error::operation_aborted`.

Parameters

expiry_time The expiry time to be used for the stream.

5.77.8 basic_socket_iostream::expires_from_now

Get the timer's expiry time relative to now.

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

Set the stream's expiry time relative to now.

```
void expires_from_now(
    const duration_type & expiry_time);
```

5.77.8.1 basic_socket_iostream::expires_from_now (1 of 2 overloads)

Get the timer's expiry time relative to now.

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

Return Value

A relative time value representing the stream's expiry time.

5.77.8.2 `basic_socket_iostream::expires_from_now (2 of 2 overloads)`

Set the stream's expiry time relative to now.

```
void expires_from_now(
    const duration_type & expiry_time);
```

This function sets the expiry time associated with the stream. Stream operations performed after this time (where the operations cannot be completed using the internal buffers) will fail with the error `asio::error::operation_aborted`.

Parameters

expiry_time The expiry time to be used for the timer.

5.77.9 `basic_socket_iostream::rdbuf`

Return a pointer to the underlying streambuf.

```
basic_socket_streambuf< Protocol, StreamSocketService, Time, TimeTraits, TimerService > * rdbuf ←
    () const;
```

5.77.10 `basic_socket_iostream::time_type`

The time type.

```
typedef TimeTraits::time_type time_type;
```

Requirements

Header: `asio/basic_socket_iostream.hpp`

Convenience header: `asio.hpp`

5.78 `basic_socket_streambuf`

Iostream streambuf for a socket.

```
template<
    typename Protocol,
    typename StreamSocketService = stream_socket_service<Protocol>,
    typename Time = boost::posix_time::ptime,
    typename TimeTraits = asio::time_traits<Time>,
    typename TimerService = deadline_timer_service<Time, TimeTraits>>
class basic_socket_streambuf :
    public basic_socket< Protocol, StreamSocketService >
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
duration_type	The duration type.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Name	Description
time_type	The time type.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_socket_streambuf	Construct a basic_socket_streambuf without establishing a connection.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the connection. Close the socket.
connect	Establish a connection. Connect the socket to the specified endpoint.
expires_at	Get the stream buffer's expiry time as an absolute time. Set the stream buffer's expiry time as an absolute time.
expires_from_now	Get the stream buffer's expiry time relative to now. Set the stream buffer's expiry time relative to now.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.

Name	Description
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.
puberror	Get the last error associated with the stream buffer.
remote_endpoint	Get the remote endpoint of the socket.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.
~basic_socket_streambuf	Destructor flushes buffered data.

Protected Member Functions

Name	Description
error	Get the last error associated with the stream buffer.
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
overflow	
setbuf	
sync	
underflow	

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

Friends

Name	Description
io_handler	
timer_handler	

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.1 basic_socket_streambuf::assign

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);

asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.78.1.1 basic_socket_streambuf::assign (1 of 2 overloads)

Inherited from basic_socket.

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

5.78.1.2 basic_socket_streambuf::assign (2 of 2 overloads)

Inherited from basic_socket.

Assign an existing native socket to the socket.

```
asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.78.2 basic_socket_streambuf::async_connect

Inherited from `basic_socket`.

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
void-or-deduced async_connect(
    const endpoint_type & peer_endpoint,
    ConnectHandler handler);
```

This function is used to asynchronously connect a socket to the specified remote endpoint. The function call always returns immediately.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected. Copies will be made of the endpoint object as required.

handler The handler to be called when the connection operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error // Result of operation
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

```
void connect_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Connect succeeded.
    }
}

...

asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.async_connect(endpoint, connect_handler);
```

5.78.3 basic_socket_streambuf::at_mark

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;  
  
bool at_mark(  
    asio::error_code & ec) const;
```

5.78.3.1 basic_socket_streambuf::at_mark (1 of 2 overloads)

Inherited from basic_socket.

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

Exceptions

asio::system_error Thrown on failure.

5.78.3.2 basic_socket_streambuf::at_mark (2 of 2 overloads)

Inherited from basic_socket.

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark(  
    asio::error_code & ec) const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

5.78.4 basic_socket_streambuf::available

Determine the number of bytes available for reading.

```
std::size_t available() const;  
  
std::size_t available(  
    asio::error_code & ec) const;
```

5.78.4.1 basic_socket_streambuf::available (1 of 2 overloads)

Inherited from basic_socket.

Determine the number of bytes available for reading.

```
std::size_t available() const;
```

This function is used to determine the number of bytes that may be read without blocking.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

Exceptions

`asio::system_error` Thrown on failure.

5.78.4.2 basic_socket_streambuf::available (2 of 2 overloads)

Inherited from basic_socket.

Determine the number of bytes available for reading.

```
std::size_t available(
    asio::error_code & ec) const;
```

This function is used to determine the number of bytes that may be read without blocking.

Parameters

`ec` Set to indicate what error occurred, if any.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

5.78.5 basic_socket_streambuf::basic_socket_streambuf

Construct a `basic_socket_streambuf` without establishing a connection.

```
basic_socket_streambuf();
```

5.78.6 basic_socket_streambuf::bind

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);
```

```
asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.78.6.1 basic_socket_streambuf::bind (1 of 2 overloads)

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
void bind(  
    const endpoint_type & endpoint);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);  
socket.open(asio::ip::tcp::v4());  
socket.bind(asio::ip::tcp::endpoint(  
    asio::ip::tcp::v4(), 12345));
```

5.78.6.2 basic_socket_streambuf::bind (2 of 2 overloads)

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
asio::error_code bind(  
    const endpoint_type & endpoint,  
    asio::error_code & ec);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);  
socket.open(asio::ip::tcp::v4());  
asio::error_code ec;  
socket.bind(asio::ip::tcp::endpoint(  
    asio::ip::tcp::v4(), 12345), ec);  
if (ec)  
{  
    // An error occurred.  
}
```

5.78.7 basic_socket_streambuf::broadcast

Inherited from socket_base.

Socket option to permit sending of broadcast messages.

```
typedef implementation_defined broadcast;
```

Implements the SOL_SOCKET/SO_BROADCAST socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.8 basic_socket_streambuf::bytes_readable

Inherited from socket_base.

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::bytes_readable command(true);
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.9 basic_socket_streambuf::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();  
  
asio::error_code cancel(  
    asio::error_code & ec);
```

5.78.9.1 basic_socket_streambuf::cancel (1 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
void cancel();
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.78.9.2 basic_socket_streambuf::cancel (2 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
asio::error_code cancel(  
    asio::error_code & ec);
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.78.10 basic_socket_streambuf::close

Close the connection.

```
basic_socket_streambuf< Protocol, StreamSocketService, Time, TimeTraits, TimerService > * close ←
();
```

Close the socket.

```
asio::error_code close(
    asio::error_code & ec);
```

5.78.10.1 basic_socket_streambuf::close (1 of 2 overloads)

Close the connection.

```
basic_socket_streambuf< Protocol, StreamSocketService, Time, TimeTraits, TimerService > * close ←
();
```

Return Value

`this` if a connection was successfully established, a null pointer otherwise.

5.78.10.2 basic_socket_streambuf::close (2 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
asio::error_code close(
    asio::error_code & ec);
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any. Note that, even if the function indicates an error, the underlying descriptor is closed.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
socket.close(ec);
if (ec)
{
    // An error occurred.
}
```

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.78.11 basic_socket_streambuf::connect

Establish a connection.

```
basic_socket_streambuf< Protocol, StreamSocketService, Time, TimeTraits, TimerService > * connect(
    const endpoint_type & endpoint);

template<
    typename T1,
    ... ,
    typename TN>
basic_socket_streambuf< Protocol, StreamSocketService > * connect(
    T1 t1,
    ... ,
    TN tn);
```

Connect the socket to the specified endpoint.

```
asio::error_code connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.78.11.1 basic_socket_streambuf::connect (1 of 3 overloads)

Establish a connection.

```
basic_socket_streambuf< Protocol, StreamSocketService, Time, TimeTraits, TimerService > * connect(
    const endpoint_type & endpoint);
```

This function establishes a connection to the specified endpoint.

Return Value

this if a connection was successfully established, a null pointer otherwise.

5.78.11.2 basic_socket_streambuf::connect (2 of 3 overloads)

Establish a connection.

```
template<
    typename T1,
    ...
    typename TN>
basic_socket_streambuf< Protocol, StreamSocketService > * connect(
    T1 t1,
    ...
    TN tn);
```

This function automatically establishes a connection based on the supplied resolver query parameters. The arguments are used to construct a resolver query object.

Return Value

this if a connection was successfully established, a null pointer otherwise.

5.78.11.3 basic_socket_streambuf::connect (3 of 3 overloads)

Inherited from *basic_socket*.

Connect the socket to the specified endpoint.

```
asio::error_code connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
asio::error_code ec;
socket.connect(endpoint, ec);
if (ec)
{
    // An error occurred.
}
```

5.78.12 basic_socket_streambuf::debug

Inherited from socket_base.

Socket option to enable socket-level debugging.

```
typedef implementation_defined debug;
```

Implements the SOL_SOCKET/SO_DEBUG socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.13 basic_socket_streambuf::do_not_route

Inherited from socket_base.

Socket option to prevent routing, use local interfaces only.

```
typedef implementation_defined do_not_route;
```

Implements the SOL_SOCKET/SO_DONTROUTE socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.14 basic_socket_streambuf::duration_type

The duration type.

```
typedef TimeTraits::duration_type duration_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.15 basic_socket_streambuf::enable_connection_aborted

Inherited from socket_base.

Socket option to report aborted connections on accept.

```
typedef implementation_defined enable_connection_aborted;
```

Implements a custom socket option that determines whether or not an accept operation is permitted to fail with asio::error::connection_aborted. By default the option is false.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.16 basic_socket_streambuf::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.17 basic_socket_streambuf::error

Get the last error associated with the stream buffer.

```
virtual const asio::error_code & error() const;
```

Return Value

An `error_code` corresponding to the last error from the stream buffer.

5.78.18 basic_socket_streambuf::expires_at

Get the stream buffer's expiry time as an absolute time.

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

Set the stream buffer's expiry time as an absolute time.

```
void expires_at(
    const time_type & expiry_time);
```

5.78.18.1 basic_socket_streambuf::expires_at (1 of 2 overloads)

Get the stream buffer's expiry time as an absolute time.

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

Return Value

An absolute time value representing the stream buffer's expiry time.

5.78.18.2 basic_socket_streambuf::expires_at (2 of 2 overloads)

Set the stream buffer's expiry time as an absolute time.

```
void expires_at(
    const time_type & expiry_time);
```

This function sets the expiry time associated with the stream. Stream operations performed after this time (where the operations cannot be completed using the internal buffers) will fail with the error `asio::error::operation_aborted`.

Parameters

expiry_time The expiry time to be used for the stream.

5.78.19 `basic_socket_streambuf::expires_from_now`

Get the stream buffer's expiry time relative to now.

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

Set the stream buffer's expiry time relative to now.

```
void expires_from_now(
    const duration_type & expiry_time);
```

5.78.19.1 `basic_socket_streambuf::expires_from_now (1 of 2 overloads)`

Get the stream buffer's expiry time relative to now.

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

Return Value

A relative time value representing the stream buffer's expiry time.

5.78.19.2 `basic_socket_streambuf::expires_from_now (2 of 2 overloads)`

Set the stream buffer's expiry time relative to now.

```
void expires_from_now(
    const duration_type & expiry_time);
```

This function sets the expiry time associated with the stream. Stream operations performed after this time (where the operations cannot be completed using the internal buffers) will fail with the error `asio::error::operation_aborted`.

Parameters

expiry_time The expiry time to be used for the timer.

5.78.20 `basic_socket_streambuf::get_implementation`

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();

const implementation_type & get_implementation() const;
```

5.78.20.1 `basic_socket_streambuf::get_implementation (1 of 2 overloads)`

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.78.20.2 `basic_socket_streambuf::get_implementation` (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.78.21 `basic_socket_streambuf::get_io_service`

Inherited from basic_io_object.

Get the `io_service` associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the `io_service` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_service` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.78.22 `basic_socket_streambuf::get_option`

Get an option from the socket.

```
void get_option(
    GettableSocketOption & option) const;

asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.78.22.1 `basic_socket_streambuf::get_option` (1 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

Exceptions

`asio::system_error` Thrown on failure.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.78.22.2 basic_socket_streambuf::get_option (2 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
asio::error_code ec;
socket.get_option(option, ec);
if (ec)
{
    // An error occurred.
}
bool is_set = option.value();
```

5.78.23 basic_socket_streambuf::get_service

Get the service associated with the I/O object.

```
service_type & get_service();

const service_type & get_service() const;
```

5.78.23.1 basic_socket_streambuf::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.78.23.2 basic_socket_streambuf::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.78.24 basic_socket_streambuf::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.78.25 basic_socket_streambuf::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.26 basic_socket_streambuf::io_control

Perform an IO control command on the socket.

```
void io_control(
    IoControlCommand & command);

asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.78.26.1 basic_socket_streambuf::io_control (1 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

5.78.26.2 basic_socket_streambuf::io_control (2 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
asio::error_code ec;
socket.io_control(command, ec);
if (ec)
{
    // An error occurred.
}
std::size_t bytes_readable = command.get();
```

5.78.27 basic_socket_streambuf::io_handler

```
friend struct io_handler();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.28 basic_socket_streambuf::is_open

Inherited from basic_socket.

Determine whether the socket is open.

```
bool is_open() const;
```

5.78.29 basic_socket_streambuf::keep_alive

Inherited from socket_base.

Socket option to send keep-alives.

```
typedef implementation_defined keep_alive;
```

Implements the SOL_SOCKET/SO_KEEPALIVE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.30 basic_socket_streambuf::linger

Inherited from socket_base.

Socket option to specify whether the socket lingers on close if unsent data is present.

```
typedef implementation_defined linger;
```

Implements the SOL_SOCKET/SO_LINGER socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option;
socket.get_option(option);
bool is_set = option.enabled();
unsigned short timeout = option.timeout();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.31 basic_socket_streambuf::local_endpoint

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;

endpoint_type local_endpoint(
    asio::error_code & ec) const;
```

5.78.31.1 basic_socket_streambuf::local_endpoint (1 of 2 overloads)

Inherited from basic_socket.

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;
```

This function is used to obtain the locally bound endpoint of the socket.

Return Value

An object that represents the local endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::endpoint endpoint = socket.local_endpoint();
```

5.78.31.2 basic_socket_streambuf::local_endpoint (2 of 2 overloads)

Inherited from basic_socket.

Get the local endpoint of the socket.

```
endpoint_type local_endpoint(
    asio::error_code & ec) const;
```

This function is used to obtain the locally bound endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the local endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.local_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.78.32 basic_socket_streambuf::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.78.32.1 basic_socket_streambuf::lowest_layer (1 of 2 overloads)

Inherited from basic_socket.

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a **basic_socket** cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.78.32.2 basic_socket_streambuf::lowest_layer (2 of 2 overloads)

Inherited from basic_socket.

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a **basic_socket** cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.78.33 basic_socket_streambuf::lowest_layer_type

Inherited from basic_socket.

A **basic_socket** is always the lowest layer.

```
typedef basic_socket< Protocol, StreamSocketService > lowest_layer_type;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.

Name	Description
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.

Name	Description
<code>basic_socket</code>	Construct a <code>basic_socket</code> without opening it. Construct and open a <code>basic_socket</code> . Construct a <code>basic_socket</code> , opening it and binding it to the given local endpoint. Construct a <code>basic_socket</code> on an existing native socket. Move-construct a <code>basic_socket</code> from another. Move-construct a <code>basic_socket</code> from a socket of another protocol type.
<code>bind</code>	Bind the socket to the given local endpoint.
<code>cancel</code>	Cancel all asynchronous operations associated with the socket.
<code>close</code>	Close the socket.
<code>connect</code>	Connect the socket to the specified endpoint.
<code>get_io_service</code>	Get the <code>io_service</code> associated with the object.
<code>get_option</code>	Get an option from the socket.
<code>io_control</code>	Perform an IO control command on the socket.
<code>is_open</code>	Determine whether the socket is open.
<code>local_endpoint</code>	Get the local endpoint of the socket.
<code>lowest_layer</code>	Get a reference to the lowest layer. Get a const reference to the lowest layer.
<code>native</code>	(Deprecated: Use <code>native_handle()</code>) Get the native socket representation.
<code>native_handle</code>	Get the native socket representation.
<code>native_non_blocking</code>	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
<code>non_blocking</code>	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
<code>open</code>	Open the socket using the specified protocol.
<code>operator=</code>	Move-assign a <code>basic_socket</code> from another. Move-assign a <code>basic_socket</code> from a socket of another protocol type.
<code>remote_endpoint</code>	Get the remote endpoint of the socket.
<code>set_option</code>	Set an option on the socket.

Name	Description
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_socket	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_socket` class template provides functionality that is common to both stream-oriented and datagram-oriented sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.34 basic_socket_streambuf::max_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.78.35 basic_socket_streambuf::message_do_not_route

Inherited from socket_base.

Specify that the data should not be subject to routing.

```
static const int message_do_not_route = implementation_defined;
```

5.78.36 basic_socket_streambuf::message_end_of_record

Inherited from socket_base.

Specifies that the data marks the end of a record.

```
static const int message_end_of_record = implementation_defined;
```

5.78.37 basic_socket_streambuf::message_flags

Inherited from socket_base.

Bitmask type for flags that can be passed to send and receive operations.

```
typedef int message_flags;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.38 basic_socket_streambuf::message_out_of_band

Inherited from socket_base.

Process out-of-band data.

```
static const int message_out_of_band = implementation_defined;
```

5.78.39 basic_socket_streambuf::message_peek

Inherited from socket_base.

Peek at incoming data without removing it from the input queue.

```
static const int message_peek = implementation_defined;
```

5.78.40 basic_socket_streambuf::native

Inherited from basic_socket.

(Deprecated: Use native_handle().) Get the native socket representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.78.41 basic_socket_streambuf::native_handle

Inherited from basic_socket.

Get the native socket representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.78.42 basic_socket_streambuf::native_handle_type

Inherited from basic_socket.

The native representation of a socket.

```
typedef StreamSocketService::native_handle_type native_handle_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.43 basic_socket_streambuf::native_non_blocking

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.78.43.1 basic_socket_streambuf::native_non_blocking (1 of 3 overloads)

Inherited from `basic_socket`.

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

This function is used to retrieve the non-blocking mode of the underlying native socket. This mode has no effect on the behaviour of the socket object's synchronous operations.

Return Value

`true` if the underlying socket is in non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Remarks

The current non-blocking mode is cached by the socket object. Consequently, the return value may be incorrect if the non-blocking mode was set directly on the native socket.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.
                errno = 0;
                int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
                ec = asio::error_code(n < 0 ? errno : 0,
                                     asio::error::get_system_category());
                total_bytes_transferred_ += ec ? 0 : n;

                // Retry operation immediately if interrupted by signal.
                if (ec == asio::error::interrupted)
                    continue;
            }
        }
    }
};
```

```

// Check if we need to run the operation again.
if (ec == asio::error::would_block
    || ec == asio::error::try_again)
{
    // We have to wait for the socket to become ready again.
    sock_.async_write_some(asio::null_buffers(), *this);
    return;
}

if (ec || n == 0)
{
    // An error occurred, or we have reached the end of the file.
    // Either way we must exit the loop so we can call the handler.
    break;
}

// Loop around to try calling sendfile again.
}
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}
};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.78.43.2 basic_socket_streambuf::native_non_blocking (2 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);
```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Exceptions

asio::system_error Thrown on failure. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.
                errno = 0;
                int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
                ec = asio::error_code(n < 0 ? errno : 0,
                                      asio::error::get_system_category());
                total_bytes_transferred_ += ec ? 0 : n;

                // Retry operation immediately if interrupted by signal.
                if (ec == asio::error::interrupted)
                    continue;

                // Check if we need to run the operation again.
                if (ec == asio::error::would_block
                    || ec == asio::error::try_again)
                {
                    // We have to wait for the socket to become ready again.
                    sock_.async_write_some(asio::null_buffers(), *this);
                    return;
                }

                if (ec || n == 0)
                {
                    // An error occurred, or we have reached the end of the file.
                    // Either way we must exit the loop so we can call the handler.
                    break;
                }
            }

            // Loop around to try calling sendfile again.
        }
    }

    // Pass result back to user's handler.
    handler_(ec, total_bytes_transferred_);
}
```

```

};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.78.43.3 basic_socket_streambuf::native_non_blocking (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);

```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

ec Set to indicate what error occurred, if any. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```

template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.

```

```

errno = 0;
int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
ec = asio::error_code(n < 0 ? errno : 0,
    asio::error::get_system_category());
total_bytes_transferred_ += ec ? 0 : n;

// Retry operation immediately if interrupted by signal.
if (ec == asio::error::interrupted)
    continue;

// Check if we need to run the operation again.
if (ec == asio::error::would_block
    || ec == asio::error::try_again)
{
    // We have to wait for the socket to become ready again.
    sock_.async_write_some(asio::null_buffers(), *this);
    return;
}

if (ec || n == 0)
{
    // An error occurred, or we have reached the end of the file.
    // Either way we must exit the loop so we can call the handler.
    break;
}

// Loop around to try calling sendfile again.
}
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}

};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.78.44 basic_socket_streambuf::native_type

Inherited from basic_socket.

(Deprecated: Use native_handle_type.) The native representation of a socket.

```
typedef StreamSocketService::native_handle_type native_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.45 basic_socket_streambuf::non_blocking

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Sets the non-blocking mode of the socket.

```
void non_blocking(  
    bool mode);  
  
asio::error_code non_blocking(  
    bool mode,  
    asio::error_code & ec);
```

5.78.45.1 basic_socket_streambuf::non_blocking (1 of 3 overloads)

Inherited from basic_socket.

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Return Value

`true` if the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.78.45.2 basic_socket_streambuf::non_blocking (2 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
void non_blocking(  
    bool mode);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.78.45.3 basic_socket_streambuf::non_blocking (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

ec Set to indicate what error occurred, if any.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.78.46 basic_socket_streambuf::non_blocking_io

Inherited from socket_base.

(Deprecated: Use `non_blocking()` IO control command to set the blocking mode of the socket.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::non_blocking_io command(true);
socket.io_control(command);
```

Requirements

Header: `asio/basic_socket_streambuf.hpp`

Convenience header: `asio.hpp`

5.78.47 basic_socket_streambuf::open

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());

asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.78.47.1 basic_socket_streambuf::open (1 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying protocol parameters to be used.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
```

5.78.47.2 basic_socket_streambuf::open (2 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying which protocol is to be used.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.78.48 basic_socket_streambuf::overflow

```
int_type overflow(
    int_type c);
```

5.78.49 basic_socket_streambuf::protocol_type

Inherited from basic_socket.

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.50 basic_socket_streambuf::puberror

Get the last error associated with the stream buffer.

```
const asio::error_code & puberror() const;
```

Return Value

An `error_code` corresponding to the last error from the stream buffer.

5.78.51 basic_socket_streambuf::receive_buffer_size

Inherited from socket_base.

Socket option for the receive buffer size of a socket.

```
typedef implementation_defined receive_buffer_size;
```

Implements the SOL_SOCKET/SO_RCVBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.52 basic_socket_streambuf::receive_low_watermark

Inherited from socket_base.

Socket option for the receive low watermark.

```
typedef implementation_defined receive_low_watermark;
```

Implements the SOL_SOCKET/SO_RCVLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.53 basic_socket_streambuf::remote_endpoint

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;

endpoint_type remote_endpoint(
    asio::error_code & ec) const;
```

5.78.53.1 basic_socket_streambuf::remote_endpoint (1 of 2 overloads)

Inherited from basic_socket.

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;
```

This function is used to obtain the remote endpoint of the socket.

Return Value

An object that represents the remote endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint();
```

5.78.53.2 basic_socket_streambuf::remote_endpoint (2 of 2 overloads)

Inherited from basic_socket.

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint(
    asio::error_code & ec) const;
```

This function is used to obtain the remote endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the remote endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.78.54 basic_socket_streambuf::reuse_address

Inherited from socket_base.

Socket option to allow the socket to be bound to an address that is already in use.

```
typedef implementation_defined reuse_address;
```

Implements the SOL_SOCKET/SO_REUSEADDR socket option.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.55 basic_socket_streambuf::send_buffer_size

Inherited from socket_base.

Socket option for the send buffer size of a socket.

```
typedef implementation_defined send_buffer_size;
```

Implements the SOL_SOCKET/SO_SNDBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.56 basic_socket_streambuf::send_low_watermark

Inherited from socket_base.

Socket option for the send low watermark.

```
typedef implementation_defined send_low_watermark;
```

Implements the SOL_SOCKET/SO SNDLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.57 basic_socket_streambuf::service

Inherited from basic_io_object.

(Deprecated: Use get_service() .) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.78.58 basic_socket_streambuf::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef StreamSocketService service_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.59 `basic_socket_streambuf::set_option`

Set an option on the socket.

```
void set_option(
    const SettableSocketOption & option);

asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.78.59.1 `basic_socket_streambuf::set_option (1 of 2 overloads)`

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.78.59.2 `basic_socket_streambuf::set_option (2 of 2 overloads)`

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

ec Set to indicate what error occurred, if any.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.78.60 basic_socket_streambuf::setbuf

```
std::streambuf * setbuf(
    char_type * s,
    std::streamsize n);
```

5.78.61 basic_socket_streambuf::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.78.61.1 basic_socket_streambuf::shutdown (1 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

Exceptions

asio::system_error Thrown on failure.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
socket.shutdown(asio::ip::tcp::socket::shutdown_send);
```

5.78.61.2 basic_socket_streambuf::shutdown (2 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

ec Set to indicate what error occurred, if any.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);
if (ec)
{
    // An error occurred.
}
```

5.78.62 basic_socket_streambuf::shutdown_type

Inherited from socket_base.

Different ways a socket may be shutdown.

```
enum shutdown_type
```

Values

shutdown_receive Shutdown the receive side of the socket.

shutdown_send Shutdown the send side of the socket.

shutdown_both Shutdown both send and receive on the socket.

5.78.63 basic_socket_streambuf::sync

```
int sync();
```

5.78.64 basic_socket_streambuf::time_type

The time type.

```
typedef TimeTraits::time_type time_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.65 basic_socket_streambuf::timer_handler

```
friend struct timer_handler();
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.78.66 basic_socket_streambuf::underflow

```
int_type underflow();
```

5.78.67 basic_socket_streambuf::~basic_socket_streambuf

Destructor flushes buffered data.

```
virtual ~basic_socket_streambuf();
```

5.79 basic_stream_socket

Provides stream-oriented socket functionality.

```
template<
    typename Protocol,
    typename StreamSocketService = stream_socket_service<Protocol>>
class basic_stream_socket :
    public basic_socket< Protocol, StreamSocketService >
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_read_some	Start an asynchronous read.
async_receive	Start an asynchronous receive.
async_send	Start an asynchronous send.
async_write_some	Start an asynchronous write.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_stream_socket	Construct a basic_stream_socket without opening it. Construct and open a basic_stream_socket. Construct a basic_stream_socket, opening it and binding it to the given local endpoint. Construct a basic_stream_socket on an existing native socket. Move-construct a basic_stream_socket from another. Move-construct a basic_stream_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.

Name	Description
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.
operator=	Move-assign a basic_stream_socket from another. Move-assign a basic_stream_socket from a socket of another protocol type.
read_some	Read some data from the socket.
receive	Receive some data on the socket. Receive some data on a connected socket.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on the socket.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.
write_some	Write some data to the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.

Name	Description
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_stream_socket` class template provides asynchronous and blocking stream-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/basic_stream_socket.hpp`

Convenience header: `asio.hpp`

5.79.1 basic_stream_socket::assign

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);

asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.79.1.1 basic_stream_socket::assign (1 of 2 overloads)

Inherited from basic_socket.

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

5.79.1.2 basic_stream_socket::assign (2 of 2 overloads)

Inherited from `basic_socket`.

Assign an existing native socket to the socket.

```
asio::error_code assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.79.2 basic_stream_socket::async_connect

Inherited from `basic_socket`.

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
void-or-deduced async_connect(
    const endpoint_type & peer_endpoint,
    ConnectHandler handler);
```

This function is used to asynchronously connect a socket to the specified remote endpoint. The function call always returns immediately.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected. Copies will be made of the endpoint object as required.

handler The handler to be called when the connection operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error // Result of operation
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Example

```
void connect_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Connect succeeded.
    }
}

...

asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.async_connect(endpoint, connect_handler);
```

5.79.3 basic_stream_socket::async_read_some

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

This function is used to asynchronously read data from the stream socket. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be read. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes read.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The read operation may not read all of the requested number of bytes. Consider using the `async_read` function if you need to ensure that the requested amount of data is read before the asynchronous operation completes.

Example

To read into a single data buffer use the `buffer` function as follows:

```
socket.async_read_some(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.4 basic_stream_socket::async_receive

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    ReadHandler handler);

template<
    typename MutableBufferSequence,
```

```
typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    ReadHandler handler);
```

5.79.4.1 basic_stream_socket::async_receive (1 of 2 overloads)

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

This function is used to asynchronously receive data from the stream socket. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The receive operation may not receive all of the requested number of bytes. Consider using the [async_read](#) function if you need to ensure that the requested amount of data is received before the asynchronous operation completes.

Example

To receive into a single data buffer use the [buffer](#) function as follows:

```
socket.async_receive(asio::buffer(data, size), handler);
```

See the [buffer](#) documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.4.2 basic_stream_socket::async_receive (2 of 2 overloads)

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    ReadHandler handler);
```

This function is used to asynchronously receive data from the stream socket. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be received. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

flags Flags specifying how the receive call is to be made.

handler The handler to be called when the receive operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes received.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The receive operation may not receive all of the requested number of bytes. Consider using the [async_read](#) function if you need to ensure that the requested amount of data is received before the asynchronous operation completes.

Example

To receive into a single data buffer use the [buffer](#) function as follows:

```
socket.async_receive(asio::buffer(data, size), 0, handler);
```

See the [buffer](#) documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.5 basic_stream_socket::async_send

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);
```

5.79.5.1 basic_stream_socket::async_send (1 of 2 overloads)

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

This function is used to asynchronously send data on the stream socket. The function call always returns immediately.

Parameters

buffers One or more data buffers to be sent on the socket. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes sent.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The send operation may not transmit all of the data to the peer. Consider using the `async_write` function if you need to ensure that all data is written before the asynchronous operation completes.

Example

To send a single data buffer use the `buffer` function as follows:

```
socket.async_send(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.5.2 basic_stream_socket::async_send (2 of 2 overloads)

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);
```

This function is used to asynchronously send data on the stream socket. The function call always returns immediately.

Parameters

buffers One or more data buffers to be sent on the socket. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

flags Flags specifying how the send call is to be made.

handler The handler to be called when the send operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes sent.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The send operation may not transmit all of the data to the peer. Consider using the `async_write` function if you need to ensure that all data is written before the asynchronous operation completes.

Example

To send a single data buffer use the `buffer` function as follows:

```
socket.async_send(asio::buffer(data, size), 0, handler);
```

See the `buffer` documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.6 basic_stream_socket::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some(
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

This function is used to asynchronously write data to the stream socket. The function call always returns immediately.

Parameters

buffers One or more data buffers to be written to the socket. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(  
    const asio::error_code& error, // Result of operation.  
    std::size_t bytes_transferred           // Number of bytes written.  
) ;
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The write operation may not transmit all of the data to the peer. Consider using the `async_write` function if you need to ensure that all data is written before the asynchronous operation completes.

Example

To write a single data buffer use the `buffer` function as follows:

```
socket.async_write_some(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.7 basic_stream_socket::at_mark

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;  
  
bool at_mark(  
    asio::error_code & ec) const;
```

5.79.7.1 basic_stream_socket::at_mark (1 of 2 overloads)

Inherited from basic_socket.

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark() const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

Exceptions

asio::system_error Thrown on failure.

5.79.7.2 basic_stream_socket::at_mark (2 of 2 overloads)

Inherited from basic_socket.

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark(
    asio::error_code & ec) const;
```

This function is used to check whether the socket input is currently positioned at the out-of-band data mark.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

A bool indicating whether the socket is at the out-of-band data mark.

5.79.8 basic_stream_socket::available

Determine the number of bytes available for reading.

```
std::size_t available() const;

std::size_t available(
    asio::error_code & ec) const;
```

5.79.8.1 basic_stream_socket::available (1 of 2 overloads)

Inherited from basic_socket.

Determine the number of bytes available for reading.

```
std::size_t available() const;
```

This function is used to determine the number of bytes that may be read without blocking.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

Exceptions

asio::system_error Thrown on failure.

5.79.8.2 basic_stream_socket::available (2 of 2 overloads)

Inherited from `basic_socket`.

Determine the number of bytes available for reading.

```
std::size_t available(  
    asio::error_code & ec) const;
```

This function is used to determine the number of bytes that may be read without blocking.

Parameters

`ec` Set to indicate what error occurred, if any.

Return Value

The number of bytes that may be read without blocking, or 0 if an error occurs.

5.79.9 basic_stream_socket::basic_stream_socket

Construct a `basic_stream_socket` without opening it.

```
explicit basic_stream_socket(  
    asio::io_service & io_service);
```

Construct and open a `basic_stream_socket`.

```
basic_stream_socket(  
    asio::io_service & io_service,  
    const protocol_type & protocol);
```

Construct a `basic_stream_socket`, opening it and binding it to the given local endpoint.

```
basic_stream_socket(  
    asio::io_service & io_service,  
    const endpoint_type & endpoint);
```

Construct a `basic_stream_socket` on an existing native socket.

```
basic_stream_socket(  
    asio::io_service & io_service,  
    const protocol_type & protocol,  
    const native_handle_type & native_socket);
```

Move-construct a `basic_stream_socket` from another.

```
basic_stream_socket(  
    basic_stream_socket && other);
```

Move-construct a `basic_stream_socket` from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename StreamSocketService1>
basic_stream_socket(
    basic_stream_socket< Protocol1, StreamSocketService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.79.9.1 basic_stream_socket::basic_stream_socket (1 of 6 overloads)

Construct a `basic_stream_socket` without opening it.

```
basic_stream_socket(
   asio::io_service & io_service);
```

This constructor creates a stream socket without opening it. The socket needs to be opened and then connected or accepted before data can be sent or received on it.

Parameters

io_service The `io_service` object that the stream socket will use to dispatch handlers for any asynchronous operations performed on the socket.

5.79.9.2 basic_stream_socket::basic_stream_socket (2 of 6 overloads)

Construct and open a `basic_stream_socket`.

```
basic_stream_socket(
    asio::io_service & io_service,
    const protocol_type & protocol);
```

This constructor creates and opens a stream socket. The socket needs to be connected or accepted before data can be sent or received on it.

Parameters

io_service The `io_service` object that the stream socket will use to dispatch handlers for any asynchronous operations performed on the socket.

protocol An object specifying protocol parameters to be used.

Exceptions

`asio::system_error` Thrown on failure.

5.79.9.3 basic_stream_socket::basic_stream_socket (3 of 6 overloads)

Construct a `basic_stream_socket`, opening it and binding it to the given local endpoint.

```
basic_stream_socket(
    asio::io_service & io_service,
    const endpoint_type & endpoint);
```

This constructor creates a stream socket and automatically opens it bound to the specified endpoint on the local machine. The protocol used is the protocol associated with the given endpoint.

Parameters

io_service The `io_service` object that the stream socket will use to dispatch handlers for any asynchronous operations performed on the socket.

endpoint An endpoint on the local machine to which the stream socket will be bound.

Exceptions

`asio::system_error` Thrown on failure.

5.79.9.4 `basic_stream_socket::basic_stream_socket (4 of 6 overloads)`

Construct a `basic_stream_socket` on an existing native socket.

```
basic_stream_socket(
    asio::io_service & io_service,
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

This constructor creates a stream socket object to hold an existing native socket.

Parameters

io_service The `io_service` object that the stream socket will use to dispatch handlers for any asynchronous operations performed on the socket.

protocol An object specifying protocol parameters to be used.

native_socket The new underlying socket implementation.

Exceptions

`asio::system_error` Thrown on failure.

5.79.9.5 `basic_stream_socket::basic_stream_socket (5 of 6 overloads)`

Move-construct a `basic_stream_socket` from another.

```
basic_stream_socket(
    basic_stream_socket && other);
```

This constructor moves a stream socket from one object to another.

Parameters

other The other `basic_stream_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_stream_socket(io_service&)` constructor.

5.79.9.6 basic_stream_socket::basic_stream_socket (6 of 6 overloads)

Move-construct a `basic_stream_socket` from a socket of another protocol type.

```
template<
    typename Protocol1,
    typename StreamSocketService1>
basic_stream_socket(
    basic_stream_socket< Protocol1, StreamSocketService1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

This constructor moves a stream socket from one object to another.

Parameters

other The other `basic_stream_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_stream_socket(io_service)` constructor.

5.79.10 basic_stream_socket::bind

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.79.10.1 basic_stream_socket::bind (1 of 2 overloads)

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

Exceptions

`asio::system_error` Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345));
```

5.79.10.2 basic_stream_socket::bind (2 of 2 overloads)

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
asio::error_code bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

This function binds the socket to the specified endpoint on the local machine.

Parameters

endpoint An endpoint on the local machine to which the socket will be bound.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
asio::error_code ec;
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345), ec);
if (ec)
{
    // An error occurred.
}
```

5.79.11 basic_stream_socket::broadcast

Inherited from socket_base.

Socket option to permit sending of broadcast messages.

```
typedef implementation_defined broadcast;
```

Implements the SOL_SOCKET/SO_BROADCAST socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.12 basic_stream_socket::bytes_readable

Inherited from socket_base.

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::bytes_readable command(true);
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.13 basic_stream_socket::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.79.13.1 basic_stream_socket::cancel (1 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
void cancel();
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the asio::error::operation_aborted error.

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.79.13.2 basic_stream_socket::cancel (2 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

Calls to `cancel()` will always fail with `asio::error::operation_not_supported` when run on Windows XP, Windows Server 2003, and earlier versions of Windows, unless `ASIO_ENABLE_CANCELIO` is defined. However, the `CancelIo` function has two issues that should be considered before enabling its use:

- It will only cancel asynchronous operations that were initiated in the current thread.
- It can appear to complete without error, but the request to cancel the unfinished operations may be silently ignored by the operating system. Whether it works or not seems to depend on the drivers that are installed.

For portable cancellation, consider using one of the following alternatives:

- Disable asio's I/O completion port backend by defining `ASIO_DISABLE_IOCP`.
- Use the `close()` function to simultaneously cancel the outstanding operations and close the socket.

When running on Windows Vista, Windows Server 2008, and later, the `CancelIoEx` function is always used. This function does not have the problems described above.

5.79.14 basic_stream_socket::close

Close the socket.

```
void close();  
  
asio::error_code close(  
    asio::error_code & ec);
```

5.79.14.1 basic_stream_socket::close (1 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
void close();
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure. Note that, even if the function indicates an error, the underlying descriptor is closed.

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.79.14.2 basic_stream_socket::close (2 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
asio::error_code close(  
    asio::error_code & ec);
```

This function is used to close the socket. Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any. Note that, even if the function indicates an error, the underlying descriptor is closed.

Example

```
asio::ip::tcp::socket socket(io_service);  
...  
asio::error_code ec;  
socket.close(ec);  
if (ec)  
{  
    // An error occurred.  
}
```

Remarks

For portable behaviour with respect to graceful closure of a connected socket, call `shutdown()` before closing the socket.

5.79.15 `basic_stream_socket::connect`

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);

asio::error_code connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.79.15.1 `basic_stream_socket::connect (1 of 2 overloads)`

Inherited from basic_socket.

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.connect(endpoint);
```

5.79.15.2 basic_stream_socket::connect (2 of 2 overloads)

Inherited from basic_socket.

Connect the socket to the specified endpoint.

```
asio::error_code connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

This function is used to connect a socket to the specified remote endpoint. The function call will block until the connection is successfully made or an error occurs.

The socket is automatically opened if it is not already open. If the connect fails, and the socket was automatically opened, the socket is not returned to the closed state.

Parameters

peer_endpoint The remote endpoint to which the socket will be connected.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
asio::error_code ec;
socket.connect(endpoint, ec);
if (ec)
{
    // An error occurred.
}
```

5.79.16 basic_stream_socket::debug

Inherited from socket_base.

Socket option to enable socket-level debugging.

```
typedef implementation_defined debug;
```

Implements the SOL_SOCKET/SO_DEBUG socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.17 basic_stream_socket::do_not_route

Inherited from socket_base.

Socket option to prevent routing, use local interfaces only.

```
typedef implementation_defined do_not_route;
```

Implements the SOL_SOCKET/SO_DONTROUTE socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.18 basic_stream_socket::enable_connection_aborted

Inherited from socket_base.

Socket option to report aborted connections on accept.

```
typedef implementation_defined enable_connection_aborted;
```

Implements a custom socket option that determines whether or not an accept operation is permitted to fail with `asio::error::connection_aborted`. By default the option is false.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.19 basic_stream_socket::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.20 basic_stream_socket::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();

const implementation_type & get_implementation() const;
```

5.79.20.1 basic_stream_socket::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.79.20.2 `basic_stream_socket::get_implementation` (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.79.21 `basic_stream_socket::get_io_service`

Inherited from basic_io_object.

Get the `io_service` associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the `io_service` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_service` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.79.22 `basic_stream_socket::get_option`

Get an option from the socket.

```
void get_option(
    GettableSocketOption & option) const;

asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.79.22.1 `basic_stream_socket::get_option` (1 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

Exceptions

`asio::system_error` Thrown on failure.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.79.22.2 basic_stream_socket::get_option (2 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

This function is used to get the current value of an option on the socket.

Parameters

option The option value to be obtained from the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the value of the SOL_SOCKET/SO_KEEPALIVE option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::keep_alive option;
asio::error_code ec;
socket.get_option(option, ec);
if (ec)
{
    // An error occurred.
}
bool is_set = option.value();
```

5.79.23 basic_stream_socket::get_service

Get the service associated with the I/O object.

```
service_type & get_service();

const service_type & get_service() const;
```

5.79.23.1 basic_stream_socket::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.79.23.2 basic_stream_socket::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.79.24 basic_stream_socket::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.79.25 basic_stream_socket::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.26 basic_stream_socket::io_control

Perform an IO control command on the socket.

```
void io_control(
    IoControlCommand & command);

asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.79.26.1 basic_stream_socket::io_control (1 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

5.79.26.2 basic_stream_socket::io_control (2 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

This function is used to execute an IO control command on the socket.

Parameters

command The IO control command to be performed on the socket.

ec Set to indicate what error occurred, if any.

Example

Getting the number of bytes ready to read:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::socket::bytes_readable command;
asio::error_code ec;
socket.io_control(command, ec);
if (ec)
{
    // An error occurred.
}
std::size_t bytes_readable = command.get();
```

5.79.27 basic_stream_socket::is_open

Inherited from basic_socket.

Determine whether the socket is open.

```
bool is_open() const;
```

5.79.28 basic_stream_socket::keep_alive

Inherited from socket_base.

Socket option to send keep-alives.

```
typedef implementation_defined keep_alive;
```

Implements the SOL_SOCKET/SO_KEEPALIVE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.29 basic_stream_socket::linger

Inherited from socket_base.

Socket option to specify whether the socket lingers on close if unsent data is present.

```
typedef implementation_defined linger;
```

Implements the SOL_SOCKET/SO_LINGER socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option;
socket.get_option(option);
bool is_set = option.enabled();
unsigned short timeout = option.timeout();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.30 basic_stream_socket::local_endpoint

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;

endpoint_type local_endpoint(
    asio::error_code & ec) const;
```

5.79.30.1 basic_stream_socket::local_endpoint (1 of 2 overloads)

Inherited from basic_socket.

Get the local endpoint of the socket.

```
endpoint_type local_endpoint() const;
```

This function is used to obtain the locally bound endpoint of the socket.

Return Value

An object that represents the local endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::endpoint endpoint = socket.local_endpoint();
```

5.79.30.2 basic_stream_socket::local_endpoint (2 of 2 overloads)

Inherited from basic_socket.

Get the local endpoint of the socket.

```
endpoint_type local_endpoint(
    asio::error_code & ec) const;
```

This function is used to obtain the locally bound endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the local endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.local_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.79.31 basic_stream_socket::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.79.31.1 basic_stream_socket::lowest_layer (1 of 2 overloads)

Inherited from basic_socket.

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a **basic_socket** cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.79.31.2 basic_stream_socket::lowest_layer (2 of 2 overloads)

Inherited from basic_socket.

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a **basic_socket** cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.79.32 basic_stream_socket::lowest_layer_type

Inherited from basic_socket.

A **basic_socket** is always the lowest layer.

```
typedef basic_socket< Protocol, StreamSocketService > lowest_layer_type;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.

Name	Description
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.

Name	Description
<code>basic_socket</code>	Construct a <code>basic_socket</code> without opening it. Construct and open a <code>basic_socket</code> . Construct a <code>basic_socket</code> , opening it and binding it to the given local endpoint. Construct a <code>basic_socket</code> on an existing native socket. Move-construct a <code>basic_socket</code> from another. Move-construct a <code>basic_socket</code> from a socket of another protocol type.
<code>bind</code>	Bind the socket to the given local endpoint.
<code>cancel</code>	Cancel all asynchronous operations associated with the socket.
<code>close</code>	Close the socket.
<code>connect</code>	Connect the socket to the specified endpoint.
<code>get_io_service</code>	Get the <code>io_service</code> associated with the object.
<code>get_option</code>	Get an option from the socket.
<code>io_control</code>	Perform an IO control command on the socket.
<code>is_open</code>	Determine whether the socket is open.
<code>local_endpoint</code>	Get the local endpoint of the socket.
<code>lowest_layer</code>	Get a reference to the lowest layer. Get a const reference to the lowest layer.
<code>native</code>	(Deprecated: Use <code>native_handle()</code>) Get the native socket representation.
<code>native_handle</code>	Get the native socket representation.
<code>native_non_blocking</code>	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
<code>non_blocking</code>	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
<code>open</code>	Open the socket using the specified protocol.
<code>operator=</code>	Move-assign a <code>basic_socket</code> from another. Move-assign a <code>basic_socket</code> from a socket of another protocol type.
<code>remote_endpoint</code>	Get the remote endpoint of the socket.
<code>set_option</code>	Set an option on the socket.

Name	Description
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_socket	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_socket` class template provides functionality that is common to both stream-oriented and datagram-oriented sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.33 basic_stream_socket::max_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.79.34 basic_stream_socket::message_do_not_route

Inherited from socket_base.

Specify that the data should not be subject to routing.

```
static const int message_do_not_route = implementation_defined;
```

5.79.35 basic_stream_socket::message_end_of_record

Inherited from socket_base.

Specifies that the data marks the end of a record.

```
static const int message_end_of_record = implementation_defined;
```

5.79.36 basic_stream_socket::message_flags

Inherited from socket_base.

Bitmask type for flags that can be passed to send and receive operations.

```
typedef int message_flags;
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.37 basic_stream_socket::message_out_of_band

Inherited from socket_base.

Process out-of-band data.

```
static const int message_out_of_band = implementation_defined;
```

5.79.38 basic_stream_socket::message_peek

Inherited from socket_base.

Peek at incoming data without removing it from the input queue.

```
static const int message_peek = implementation_defined;
```

5.79.39 basic_stream_socket::native

Inherited from basic_socket.

(Deprecated: Use native_handle().) Get the native socket representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.79.40 basic_stream_socket::native_handle

Inherited from basic_socket.

Get the native socket representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the socket. This is intended to allow access to native socket functionality that is not otherwise provided.

5.79.41 basic_stream_socket::native_handle_type

The native representation of a socket.

```
typedef StreamSocketService::native_handle_type native_handle_type;
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.42 basic_stream_socket::native_non_blocking

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.79.42.1 basic_stream_socket::native_non_blocking (1 of 3 overloads)

Inherited from `basic_socket`.

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking() const;
```

This function is used to retrieve the non-blocking mode of the underlying native socket. This mode has no effect on the behaviour of the socket object's synchronous operations.

Return Value

`true` if the underlying socket is in non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Remarks

The current non-blocking mode is cached by the socket object. Consequently, the return value may be incorrect if the non-blocking mode was set directly on the native socket.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.
                errno = 0;
                int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
                ec = asio::error_code(n < 0 ? errno : 0,
                                     asio::error::get_system_category());
                total_bytes_transferred_ += ec ? 0 : n;

                // Retry operation immediately if interrupted by signal.
                if (ec == asio::error::interrupted)
                    continue;
            }
        }
    }
};
```

```

// Check if we need to run the operation again.
if (ec == asio::error::would_block
    || ec == asio::error::try_again)
{
    // We have to wait for the socket to become ready again.
    sock_.async_write_some(asio::null_buffers(), *this);
    return;
}

if (ec || n == 0)
{
    // An error occurred, or we have reached the end of the file.
    // Either way we must exit the loop so we can call the handler.
    break;
}

// Loop around to try calling sendfile again.
}
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}
};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.79.42.2 basic_stream_socket::native_non_blocking (2 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```
void native_non_blocking(
    bool mode);
```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Exceptions

asio::system_error Thrown on failure. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```
template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.
                errno = 0;
                int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
                ec = asio::error_code(n < 0 ? errno : 0,
                                      asio::error::get_system_category());
                total_bytes_transferred_ += ec ? 0 : n;

                // Retry operation immediately if interrupted by signal.
                if (ec == asio::error::interrupted)
                    continue;

                // Check if we need to run the operation again.
                if (ec == asio::error::would_block
                    || ec == asio::error::try_again)
                {
                    // We have to wait for the socket to become ready again.
                    sock_.async_write_some(asio::null_buffers(), *this);
                    return;
                }

                if (ec || n == 0)
                {
                    // An error occurred, or we have reached the end of the file.
                    // Either way we must exit the loop so we can call the handler.
                    break;
                }
            }

            // Loop around to try calling sendfile again.
        }

        // Pass result back to user's handler.
        handler_(ec, total_bytes_transferred_);
    }
}
```

```

};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.79.42.3 basic_stream_socket::native_non_blocking (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);

```

This function is used to modify the non-blocking mode of the underlying native socket. It has no effect on the behaviour of the socket object's synchronous operations.

Parameters

mode If `true`, the underlying socket is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

ec Set to indicate what error occurred, if any. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

Example

This function is intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The following example illustrates how Linux's `sendfile` system call might be encapsulated:

```

template <typename Handler>
struct sendfile_op
{
    tcp::socket& sock_;
    int fd_;
    Handler handler_;
    off_t offset_;
    std::size_t total_bytes_transferred_;

    // Function call operator meeting WriteHandler requirements.
    // Used as the handler for the async_write_some operation.
    void operator()(asio::error_code ec, std::size_t)
    {
        // Put the underlying socket into non-blocking mode.
        if (!ec)
            if (!sock_.native_non_blocking())
                sock_.native_non_blocking(true, ec);

        if (!ec)
        {
            for (;;)
            {
                // Try the system call.

```

```

errno = 0;
int n = ::sendfile(sock_.native_handle(), fd_, &offset_, 65536);
ec = asio::error_code(n < 0 ? errno : 0,
    asio::error::get_system_category());
total_bytes_transferred_ += ec ? 0 : n;

// Retry operation immediately if interrupted by signal.
if (ec == asio::error::interrupted)
    continue;

// Check if we need to run the operation again.
if (ec == asio::error::would_block
    || ec == asio::error::try_again)
{
    // We have to wait for the socket to become ready again.
    sock_.async_write_some(asio::null_buffers(), *this);
    return;
}

if (ec || n == 0)
{
    // An error occurred, or we have reached the end of the file.
    // Either way we must exit the loop so we can call the handler.
    break;
}

// Loop around to try calling sendfile again.
}
}

// Pass result back to user's handler.
handler_(ec, total_bytes_transferred_);
}

};

template <typename Handler>
void async_sendfile(tcp::socket& sock, int fd, Handler h)
{
    sendfile_op<Handler> op = { sock, fd, h, 0, 0 };
    sock.async_write_some(asio::null_buffers(), op);
}

```

5.79.43 basic_stream_socket::native_type

(Deprecated: Use native_handle_type.) The native representation of a socket.

```
typedef StreamSocketService::native_handle_type native_type;
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.44 basic_stream_socket::non_blocking

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Sets the non-blocking mode of the socket.

```
void non_blocking(
    bool mode);

asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.79.44.1 basic_stream_socket::non_blocking (1 of 3 overloads)

Inherited from basic_socket.

Gets the non-blocking mode of the socket.

```
bool non_blocking() const;
```

Return Value

`true` if the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.79.44.2 basic_stream_socket::non_blocking (2 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
void non_blocking(
    bool mode);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.79.44.3 basic_stream_socket::non_blocking (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

Parameters

mode If `true`, the socket's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

ec Set to indicate what error occurred, if any.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.79.45 basic_stream_socket::non_blocking_io

Inherited from socket_base.

(Deprecated: Use `non_blocking()` IO control command to set the blocking mode of the socket.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::non_blocking_io command(true);
socket.io_control(command);
```

Requirements

Header: `asio/basic_stream_socket.hpp`

Convenience header: `asio.hpp`

5.79.46 basic_stream_socket::open

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());

asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.79.46.1 basic_stream_socket::open (1 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying protocol parameters to be used.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
socket.open(asio::ip::tcp::v4());
```

5.79.46.2 basic_stream_socket::open (2 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
asio::error_code open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

This function opens the socket so that it will use the specified protocol.

Parameters

protocol An object specifying which protocol is to be used.

ec Set to indicate what error occurred, if any.

Example

```
asio::ip::tcp::socket socket(io_service);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.79.47 basic_stream_socket::operator=

Move-assign a `basic_stream_socket` from another.

```
basic_stream_socket & operator=(  
    basic_stream_socket && other);
```

Move-assign a `basic_stream_socket` from a socket of another protocol type.

```
template<  
    typename Protocol1,  
    typename StreamSocketService1>  
enable_if< is_convertible< Protocol1, Protocol >::value, basic_stream_socket >::type & operator <=  
=(  
    basic_stream_socket< Protocol1, StreamSocketService1 > && other);
```

5.79.47.1 basic_stream_socket::operator= (1 of 2 overloads)

Move-assign a `basic_stream_socket` from another.

```
basic_stream_socket & operator=(  
    basic_stream_socket && other);
```

This assignment operator moves a stream socket from one object to another.

Parameters

other The other `basic_stream_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_stream_socket(io_service&)` constructor.

5.79.47.2 basic_stream_socket::operator= (2 of 2 overloads)

Move-assign a `basic_stream_socket` from a socket of another protocol type.

```
template<  
    typename Protocol1,  
    typename StreamSocketService1>  
enable_if< is_convertible< Protocol1, Protocol >::value, basic_stream_socket >::type & operator <=  
=(  
    basic_stream_socket< Protocol1, StreamSocketService1 > && other);
```

This assignment operator moves a stream socket from one object to another.

Parameters

other The other `basic_stream_socket` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_stream_socket(io_service&)` constructor.

5.79.48 basic_stream_socket::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.49 basic_stream_socket::read_some

Read some data from the socket.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);

template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.79.49.1 basic_stream_socket::read_some (1 of 2 overloads)

Read some data from the socket.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);
```

This function is used to read data from the stream socket. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be read.

Return Value

The number of bytes read.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The `read_some` operation may not read all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

Example

To read into a single data buffer use the `buffer` function as follows:

```
socket.read_some(asio::buffer(data, size));
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.49.2 basic_stream_socket::read_some (2 of 2 overloads)

Read some data from the socket.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to read data from the stream socket. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be read.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes read. Returns 0 if an error occurred.

Remarks

The `read_some` operation may not read all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

5.79.50 basic_stream_socket::receive

Receive some data on the socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers);

template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags);
```

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.79.50.1 basic_stream_socket::receive (1 of 3 overloads)

Receive some data on the socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers);
```

This function is used to receive data on the stream socket. The function call will block until one or more bytes of data has been received successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

Return Value

The number of bytes received.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The receive operation may not receive all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

Example

To receive into a single data buffer use the `buffer` function as follows:

```
socket.receive(asio::buffer(data, size));
```

See the `buffer` documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.50.2 basic_stream_socket::receive (2 of 3 overloads)

Receive some data on the socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags);
```

This function is used to receive data on the stream socket. The function call will block until one or more bytes of data has been received successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

flags Flags specifying how the receive call is to be made.

Return Value

The number of bytes received.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The receive operation may not receive all of the requested number of bytes. Consider using the [read](#) function if you need to ensure that the requested amount of data is read before the blocking operation completes.

Example

To receive into a single data buffer use the [buffer](#) function as follows:

```
socket.receive(asio::buffer(data, size), 0);
```

See the [buffer](#) documentation for information on receiving into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.50.3 basic_stream_socket::receive (3 of 3 overloads)

Receive some data on a connected socket.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to receive data on the stream socket. The function call will block until one or more bytes of data has been received successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be received.

flags Flags specifying how the receive call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes received. Returns 0 if an error occurred.

Remarks

The receive operation may not receive all of the requested number of bytes. Consider using the [read](#) function if you need to ensure that the requested amount of data is read before the blocking operation completes.

5.79.51 basic_stream_socket::receive_buffer_size

Inherited from socket_base.

Socket option for the receive buffer size of a socket.

```
typedef implementation_defined receive_buffer_size;
```

Implements the SOL_SOCKET/SO_RCVBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.52 basic_stream_socket::receive_low_watermark

Inherited from socket_base.

Socket option for the receive low watermark.

```
typedef implementation_defined receive_low_watermark;
```

Implements the SOL_SOCKET/SO_RCVLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.53 basic_stream_socket::remote_endpoint

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;

endpoint_type remote_endpoint(
    asio::error_code & ec) const;
```

5.79.53.1 basic_stream_socket::remote_endpoint (1 of 2 overloads)

Inherited from basic_socket.

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint() const;
```

This function is used to obtain the remote endpoint of the socket.

Return Value

An object that represents the remote endpoint of the socket.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint();
```

5.79.53.2 basic_stream_socket::remote_endpoint (2 of 2 overloads)

Inherited from basic_socket.

Get the remote endpoint of the socket.

```
endpoint_type remote_endpoint(
    asio::error_code & ec) const;
```

This function is used to obtain the remote endpoint of the socket.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

An object that represents the remote endpoint of the socket. Returns a default-constructed endpoint object if an error occurred.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.79.54 basic_stream_socket::reuse_address

Inherited from socket_base.

Socket option to allow the socket to be bound to an address that is already in use.

```
typedef implementation_defined reuse_address;
```

Implements the SOL_SOCKET/SO_REUSEADDR socket option.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.55 basic_stream_socket::send

Send some data on the socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags);

template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.79.55.1 basic_stream_socket::send (1 of 3 overloads)

Send some data on the socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers);
```

This function is used to send data on the stream socket. The function call will block until one or more bytes of the data has been sent successfully, or an until error occurs.

Parameters

buffers One or more data buffers to be sent on the socket.

Return Value

The number of bytes sent.

Exceptions

asio::system_error Thrown on failure.

Remarks

The send operation may not transmit all of the data to the peer. Consider using the `write` function if you need to ensure that all data is written before the blocking operation completes.

Example

To send a single data buffer use the `buffer` function as follows:

```
socket.send(asio::buffer(data, size));
```

See the `buffer` documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.55.2 basic_stream_socket::send (2 of 3 overloads)

Send some data on the socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags);
```

This function is used to send data on the stream socket. The function call will block until one or more bytes of the data has been sent successfully, or an until error occurs.

Parameters

buffers One or more data buffers to be sent on the socket.

flags Flags specifying how the send call is to be made.

Return Value

The number of bytes sent.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The send operation may not transmit all of the data to the peer. Consider using the `write` function if you need to ensure that all data is written before the blocking operation completes.

Example

To send a single data buffer use the `buffer` function as follows:

```
socket.send(asio::buffer(data, size), 0);
```

See the `buffer` documentation for information on sending multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.79.55.3 basic_stream_socket::send (3 of 3 overloads)

Send some data on the socket.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

This function is used to send data on the stream socket. The function call will block until one or more bytes of the data has been sent successfully, or an until error occurs.

Parameters

buffers One or more data buffers to be sent on the socket.

flags Flags specifying how the send call is to be made.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes sent. Returns 0 if an error occurred.

Remarks

The send operation may not transmit all of the data to the peer. Consider using the [write](#) function if you need to ensure that all data is written before the blocking operation completes.

5.79.56 basic_stream_socket::send_buffer_size

Inherited from socket_base.

Socket option for the send buffer size of a socket.

```
typedef implementation_defined send_buffer_size;
```

Implements the SOL_SOCKET/SO_SNDBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.57 basic_stream_socket::send_low_watermark

Inherited from socket_base.

Socket option for the send low watermark.

```
typedef implementation_defined send_low_watermark;
```

Implements the SOL_SOCKET/SO SNDLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.58 basic_stream_socket::service

Inherited from basic_io_object.

(Deprecated: Use get_service().) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.79.59 basic_stream_socket::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef StreamSocketService service_type;
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.79.60 basic_stream_socket::set_option

Set an option on the socket.

```
void set_option(
    const SettableSocketOption & option);

asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.79.60.1 basic_stream_socket::set_option (1 of 2 overloads)

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

Exceptions

asio::system_error Thrown on failure.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.79.60.2 basic_stream_socket::set_option (2 of 2 overloads)

Inherited from `basic_socket`.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

This function is used to set an option on the socket.

Parameters

option The new option value to be set on the socket.

ec Set to indicate what error occurred, if any.

Example

Setting the IPPROTO_TCP/TCP_NODELAY option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.79.61 basic_stream_socket::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.79.61.1 basic_stream_socket::shutdown (1 of 2 overloads)

Inherited from `basic_socket`.

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

Exceptions

asio::system_error Thrown on failure.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
socket.shutdown(asio::ip::tcp::socket::shutdown_send);
```

5.79.61.2 basic_stream_socket::shutdown (2 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
asio::error_code shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

This function is used to disable send operations, receive operations, or both.

Parameters

what Determines what types of operation will no longer be allowed.

ec Set to indicate what error occurred, if any.

Example

Shutting down the send side of the socket:

```
asio::ip::tcp::socket socket(io_service);
...
asio::error_code ec;
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);
if (ec)
{
    // An error occurred.
}
```

5.79.62 basic_stream_socket::shutdown_type

Inherited from socket_base.

Different ways a socket may be shutdown.

```
enum shutdown_type
```

Values

shutdown_receive Shutdown the receive side of the socket.

shutdown_send Shutdown the send side of the socket.

shutdown_both Shutdown both send and receive on the socket.

5.79.63 basic_stream_socket::write_some

Write some data to the socket.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.79.63.1 basic_stream_socket::write_some (1 of 2 overloads)

Write some data to the socket.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

This function is used to write data to the stream socket. The function call will block until one or more bytes of the data has been written successfully, or until an error occurs.

Parameters

buffers One or more data buffers to be written to the socket.

Return Value

The number of bytes written.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The `write_some` operation may not transmit all of the data to the peer. Consider using the `write` function if you need to ensure that all data is written before the blocking operation completes.

Example

To write a single data buffer use the **buffer** function as follows:

```
socket.write_some(asio::buffer(data, size));
```

See the **buffer** documentation for information on writing multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

5.79.63.2 basic_stream_socket::write_some (2 of 2 overloads)

Write some data to the socket.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to write data to the stream socket. The function call will block until one or more bytes of the data has been written successfully, or until an error occurs.

Parameters

buffers One or more data buffers to be written to the socket.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes written. Returns 0 if an error occurred.

Remarks

The write_some operation may not transmit all of the data to the peer. Consider using the **write** function if you need to ensure that all data is written before the blocking operation completes.

5.80 basic_streambuf

Automatically resizable buffer class based on std::streambuf.

```
template<
    typename Allocator = std::allocator<char>>
class basic_streambuf :
    noncopyable
```

Types

Name	Description
const_buffers_type	The type used to represent the input sequence as a list of buffers.
mutable_buffers_type	The type used to represent the output sequence as a list of buffers.

Member Functions

Name	Description
<code>basic_streambuf</code>	Construct a <code>basic_streambuf</code> object.
<code>commit</code>	Move characters from the output sequence to the input sequence.
<code>consume</code>	Remove characters from the input sequence.
<code>data</code>	Get a list of buffers that represents the input sequence.
<code>max_size</code>	Get the maximum size of the <code>basic_streambuf</code> .
<code>prepare</code>	Get a list of buffers that represents the output sequence, with the given size.
<code>size</code>	Get the size of the input sequence.

Protected Member Functions

Name	Description
<code>overflow</code>	Override <code>std::streambuf</code> behaviour.
<code>reserve</code>	
<code>underflow</code>	Override <code>std::streambuf</code> behaviour.

The `basic_streambuf` class is derived from `std::streambuf` to associate the `streambuf`'s input and output sequences with one or more character arrays. These character arrays are internal to the `basic_streambuf` object, but direct access to the array elements is provided to permit them to be used efficiently with I/O operations. Characters written to the output sequence of a `basic_streambuf` object are appended to the input sequence of the same object.

The `basic_streambuf` class's public interface is intended to permit the following implementation strategies:

- A single contiguous character array, which is reallocated as necessary to accommodate changes in the size of the character sequence. This is the implementation approach currently used in Asio.
- 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_streambuf` 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 `basic_streambuf` object, the following invariant holds:

```
size() <= max_size()
```

Any member function that would, if successful, cause the invariant to be violated shall throw an exception of class `std::length_error`.

The constructor for `basic_streambuf` 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 `basic_streambuf` object.

Examples

Writing directly from an streambuf to a socket:

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

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

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

Reading from a socket directly into a streambuf:

```
asio::streambuf b;

// reserve 512 bytes in output sequence
asio::streambuf::mutable_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;
```

Requirements

Header: asio/basic_streambuf.hpp

Convenience header: asio.hpp

5.80.1 basic_streambuf::basic_streambuf

Construct a **basic_streambuf** object.

```
basic_streambuf(
    std::size_t maximum_size = (std::numeric_limits< std::size_t >::max)(),
    const Allocator & allocator = Allocator());
```

Constructs a streambuf with the specified maximum size. The initial size of the streambuf's input sequence is 0.

5.80.2 basic_streambuf::commit

Move characters from the output sequence to the input sequence.

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

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.

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

Remarks

If n is greater than the size of the output sequence, the entire output sequence is moved to the input sequence and no error is issued.

5.80.3 `basic_streambuf::const_buffers_type`

The type used to represent the input sequence as a list of buffers.

```
typedef implementation_defined const_buffers_type;
```

Requirements

Header: asio/basic_streambuf.hpp

Convenience header: asio.hpp

5.80.4 `basic_streambuf::consume`

Remove characters from the input sequence.

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

Removes n characters from the beginning of the input sequence.

Remarks

If n is greater than the size of the input sequence, the entire input sequence is consumed and no error is issued.

5.80.5 `basic_streambuf::data`

Get a list of buffers that represents the input sequence.

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

Return Value

An object of type `const_buffers_type` that satisfies `ConstBufferSequence` requirements, representing all character arrays in the input sequence.

Remarks

The returned object is invalidated by any `basic_streambuf` member function that modifies the input sequence or output sequence.

5.80.6 `basic_streambuf::max_size`

Get the maximum size of the `basic_streambuf`.

```
std::size_t max_size() const;
```

Return Value

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

5.80.7 `basic_streambuf::mutable_buffers_type`

The type used to represent the output sequence as a list of buffers.

```
typedef implementation_defined mutable_buffers_type;
```

Requirements

Header: asio/basic_streambuf.hpp

Convenience header: asio.hpp

5.80.8 `basic_streambuf::overflow`

Override std::streambuf behaviour.

```
int_type overflow(
    int_type c);
```

Behaves according to the specification of `std::streambuf::overflow()`, with the specialisation that `std::length_error` is thrown if appending the character to the input sequence would require the condition `size() > max_size()` to be true.

5.80.9 `basic_streambuf::prepare`

Get a list of buffers that represents the output sequence, with the given size.

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

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

Return Value

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.

Exceptions

`std::length_error` If `size() + n > max_size()`.

Remarks

The returned object is invalidated by any `basic_streambuf` member function that modifies the input sequence or output sequence.

5.80.10 `basic_streambuf::reserve`

```
void reserve(
    std::size_t n);
```

5.80.11 basic_streambuf::size

Get the size of the input sequence.

```
std::size_t size() const;
```

Return Value

The size of the input sequence. The value is 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);
}
```

5.80.12 basic_streambuf::underflow

Override std::streambuf behaviour.

```
int_type underflow();
```

Behaves according to the specification of std::streambuf::underflow().

5.81 basic_waitable_timer

Provides waitable timer functionality.

```
template<
    typename Clock,
    typename WaitTraits = asio::wait_traits<Clock>,
    typename WaitableTimerService = waitable_timer_service<Clock, WaitTraits>>
class basic_waitable_timer :
    public basic_io_object<WaitableTimerService>
```

Types

Name	Description
clock_type	The clock type.
duration	The duration type of the clock.
implementation_type	The underlying implementation type of I/O object.
service_type	The type of the service that will be used to provide I/O operations.
time_point	The time point type of the clock.
traits_type	The wait traits type.

Member Functions

Name	Description
<code>async_wait</code>	Start an asynchronous wait on the timer.
<code>basic_writable_timer</code>	Constructor. Constructor to set a particular expiry time as an absolute time. Constructor to set a particular expiry time relative to now.
<code>cancel</code>	Cancel any asynchronous operations that are waiting on the timer.
<code>cancel_one</code>	Cancels one asynchronous operation that is waiting on the timer.
<code>expires_at</code>	Get the timer's expiry time as an absolute time. Set the timer's expiry time as an absolute time.
<code>expires_from_now</code>	Get the timer's expiry time relative to now. Set the timer's expiry time relative to now.
<code>get_io_service</code>	Get the io_service associated with the object.
<code>wait</code>	Perform a blocking wait on the timer.

Protected Member Functions

Name	Description
<code>get_implementation</code>	Get the underlying implementation of the I/O object.
<code>get_service</code>	Get the service associated with the I/O object.

Protected Data Members

Name	Description
<code>implementation</code>	(Deprecated: Use <code>get_implementation()</code>) The underlying implementation of the I/O object.
<code>service</code>	(Deprecated: Use <code>get_service()</code>) The service associated with the I/O object.

The `basic_writable_timer` class template provides the ability to perform a blocking or asynchronous wait for a timer to expire.

A writable timer is always in one of two states: "expired" or "not expired". If the `wait()` or `async_wait()` function is called on an expired timer, the wait operation will complete immediately.

Most applications will use one of the `steady_timer`, `system_timer` or `high_resolution_timer` typedefs.

Remarks

This waitable timer functionality is for use with the C++11 standard library's `<chrono>` facility, or with the Boost.Chrono library.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Examples

Performing a blocking wait (C++11):

```
// Construct a timer without setting an expiry time.  
asio::steady_timer timer(io_service);  
  
// Set an expiry time relative to now.  
timer.expires_from_now(std::chrono::seconds(5));  
  
// Wait for the timer to expire.  
timer.wait();
```

Performing an asynchronous wait (C++11):

```
void handler(const asio::error_code& error)  
{  
    if (!error)  
    {  
        // Timer expired.  
    }  
}  
  
...  
  
// Construct a timer with an absolute expiry time.  
asio::steady_timer timer(io_service,  
    std::chrono::steady_clock::now() + std::chrono::seconds(60));  
  
// Start an asynchronous wait.  
timer.async_wait(handler);
```

Changing an active waitable timer's expiry time

Changing the expiry time of a timer while there are pending asynchronous waits causes those wait operations to be cancelled. To ensure that the action associated with the timer is performed only once, use something like this: used:

```
void on_some_event()  
{  
    if (my_timer.expires_from_now(seconds(5)) > 0)  
    {  
        // We managed to cancel the timer. Start new asynchronous wait.  
        my_timer.async_wait(on_timeout);  
    }  
    else  
    {  
        // Too late, timer has already expired!  
    }  
}
```

```

void on_timeout(const asio::error_code& e)
{
    if (e != asio::error::operation_aborted)
    {
        // Timer was not cancelled, take necessary action.
    }
}

```

- The `asio::basic_waitable_timer::expires_from_now()` function cancels any pending asynchronous waits, and returns the number of asynchronous waits that were cancelled. If it returns 0 then you were too late and the wait handler has already been executed, or will soon be executed. If it returns 1 then the wait handler was successfully cancelled.
- If a wait handler is cancelled, the `error_code` passed to it contains the value `asio::error::operation_aborted`.

Requirements

Header: `asio/basic_waitable_timer.hpp`

Convenience header: `asio.hpp`

5.81.1 `basic_waitable_timer::async_wait`

Start an asynchronous wait on the timer.

```

template<
    typename WaitHandler>
void-or-deduced async_wait(
    WaitHandler handler);

```

This function may be used to initiate an asynchronous wait against the timer. It always returns immediately.

For each call to `async_wait()`, the supplied handler will be called exactly once. The handler will be called when:

- The timer has expired.
- The timer was cancelled, in which case the handler is passed the error code `asio::error::operation_aborted`.

Parameters

handler The handler to be called when the timer expires. Copies will be made of the handler as required. The function signature of the handler must be:

```

void handler(
    const asio::error_code& error // Result of operation.
);

```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.81.2 basic_waitable_timer::basic_waitable_timer

Constructor.

```
explicit basic_waitable_timer(
    asio::io_service & io_service);
```

Constructor to set a particular expiry time as an absolute time.

```
basic_waitable_timer(
    asio::io_service & io_service,
    const time_point & expiry_time);
```

Constructor to set a particular expiry time relative to now.

```
basic_waitable_timer(
    asio::io_service & io_service,
    const duration & expiry_time);
```

5.81.2.1 basic_waitable_timer::basic_waitable_timer (1 of 3 overloads)

Constructor.

```
basic_waitable_timer(
    asio::io_service & io_service);
```

This constructor creates a timer without setting an expiry time. The `expires_at()` or `expires_from_now()` functions must be called to set an expiry time before the timer can be waited on.

Parameters

io_service The `io_service` object that the timer will use to dispatch handlers for any asynchronous operations performed on the timer.

5.81.2.2 basic_waitable_timer::basic_waitable_timer (2 of 3 overloads)

Constructor to set a particular expiry time as an absolute time.

```
basic_waitable_timer(
    asio::io_service & io_service,
    const time_point & expiry_time);
```

This constructor creates a timer and sets the expiry time.

Parameters

io_service The `io_service` object that the timer will use to dispatch handlers for any asynchronous operations performed on the timer.

expiry_time The expiry time to be used for the timer, expressed as an absolute time.

5.81.2.3 basic_waitable_timer::basic_waitable_timer (3 of 3 overloads)

Constructor to set a particular expiry time relative to now.

```
basic_waitable_timer(
    asio::io_service & io_service,
    const duration & expiry_time);
```

This constructor creates a timer and sets the expiry time.

Parameters

io_service The `io_service` object that the timer will use to dispatch handlers for any asynchronous operations performed on the timer.
expiry_time The expiry time to be used for the timer, relative to now.

5.81.3 `basic_waitable_timer::cancel`

Cancel any asynchronous operations that are waiting on the timer.

```
std::size_t cancel();  
  
std::size_t cancel(  
    asio::error_code & ec);
```

5.81.3.1 `basic_waitable_timer::cancel (1 of 2 overloads)`

Cancel any asynchronous operations that are waiting on the timer.

```
std::size_t cancel();
```

This function forces the completion of any pending asynchronous wait operations against the timer. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Cancelling the timer does not change the expiry time.

Return Value

The number of asynchronous operations that were cancelled.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

If the timer has already expired when `cancel()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.81.3.2 `basic_waitable_timer::cancel (2 of 2 overloads)`

Cancel any asynchronous operations that are waiting on the timer.

```
std::size_t cancel(  
    asio::error_code & ec);
```

This function forces the completion of any pending asynchronous wait operations against the timer. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Cancelling the timer does not change the expiry time.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of asynchronous operations that were cancelled.

Remarks

If the timer has already expired when `cancel()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.81.4 `basic_waitable_timer::cancel_one`

Cancels one asynchronous operation that is waiting on the timer.

```
std::size_t cancel_one();  
  
std::size_t cancel_one(  
    asio::error_code & ec);
```

5.81.4.1 `basic_waitable_timer::cancel_one (1 of 2 overloads)`

Cancels one asynchronous operation that is waiting on the timer.

```
std::size_t cancel_one();
```

This function forces the completion of one pending asynchronous wait operation against the timer. Handlers are cancelled in FIFO order. The handler for the cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Cancelling the timer does not change the expiry time.

Return Value

The number of asynchronous operations that were cancelled. That is, either 0 or 1.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

If the timer has already expired when `cancel_one()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.81.4.2 `basic_waitable_timer::cancel_one` (2 of 2 overloads)

Cancels one asynchronous operation that is waiting on the timer.

```
std::size_t cancel_one(  
    asio::error_code & ec);
```

This function forces the completion of one pending asynchronous wait operation against the timer. Handlers are cancelled in FIFO order. The handler for the cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Cancelling the timer does not change the expiry time.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of asynchronous operations that were cancelled. That is, either 0 or 1.

Remarks

If the timer has already expired when `cancel_one()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.81.5 `basic_waitable_timer::clock_type`

The clock type.

```
typedef Clock clock_type;
```

Requirements

Header: `asio/basic_waitable_timer.hpp`

Convenience header: `asio.hpp`

5.81.6 basic_waitable_timer::duration

The duration type of the clock.

```
typedef clock_type::duration duration;
```

Requirements

Header: asio/basic_waitable_timer.hpp

Convenience header: asio.hpp

5.81.7 basic_waitable_timer::expires_at

Get the timer's expiry time as an absolute time.

```
time_point expires_at() const;
```

Set the timer's expiry time as an absolute time.

```
std::size_t expires_at(
    const time_point & expiry_time);

std::size_t expires_at(
    const time_point & expiry_time,
    asio::error_code & ec);
```

5.81.7.1 basic_waitable_timer::expires_at (1 of 3 overloads)

Get the timer's expiry time as an absolute time.

```
time_point expires_at() const;
```

This function may be used to obtain the timer's current expiry time. Whether the timer has expired or not does not affect this value.

5.81.7.2 basic_waitable_timer::expires_at (2 of 3 overloads)

Set the timer's expiry time as an absolute time.

```
std::size_t expires_at(
    const time_point & expiry_time);
```

This function sets the expiry time. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Parameters

expiry_time The expiry time to be used for the timer.

Return Value

The number of asynchronous operations that were cancelled.

Exceptions

asio::system_error Thrown on failure.

Remarks

If the timer has already expired when `expires_at()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.81.7.3 `basic_waitable_timer::expires_at` (3 of 3 overloads)

Set the timer's expiry time as an absolute time.

```
std::size_t expires_at(
    const time_point & expiry_time,
    asio::error_code & ec);
```

This function sets the expiry time. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Parameters

expiry_time The expiry time to be used for the timer.

ec Set to indicate what error occurred, if any.

Return Value

The number of asynchronous operations that were cancelled.

Remarks

If the timer has already expired when `expires_at()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.81.8 `basic_waitable_timer::expires_from_now`

Get the timer's expiry time relative to now.

```
duration expires_from_now() const;
```

Set the timer's expiry time relative to now.

```
std::size_t expires_from_now(
    const duration & expiry_time);

std::size_t expires_from_now(
    const duration & expiry_time,
    asio::error_code & ec);
```

5.81.8.1 `basic_waitable_timer::expires_from_now (1 of 3 overloads)`

Get the timer's expiry time relative to now.

```
duration expires_from_now() const;
```

This function may be used to obtain the timer's current expiry time. Whether the timer has expired or not does not affect this value.

5.81.8.2 `basic_waitable_timer::expires_from_now (2 of 3 overloads)`

Set the timer's expiry time relative to now.

```
std::size_t expires_from_now(
    const duration & expiry_time);
```

This function sets the expiry time. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Parameters

expiry_time The expiry time to be used for the timer.

Return Value

The number of asynchronous operations that were cancelled.

Exceptions

asio::system_error Thrown on failure.

Remarks

If the timer has already expired when `expires_from_now()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.81.8.3 `basic_waitable_timer::expires_from_now` (3 of 3 overloads)

Set the timer's expiry time relative to now.

```
std::size_t expires_from_now(
    const duration & expiry_time,
    asio::error_code & ec);
```

This function sets the expiry time. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `asio::error::operation_aborted` error code.

Parameters

expiry_time The expiry time to be used for the timer.

ec Set to indicate what error occurred, if any.

Return Value

The number of asynchronous operations that were cancelled.

Remarks

If the timer has already expired when `expires_from_now()` is called, then the handlers for asynchronous wait operations will:

- have already been invoked; or
- have been queued for invocation in the near future.

These handlers can no longer be cancelled, and therefore are passed an error code that indicates the successful completion of the wait operation.

5.81.9 `basic_waitable_timer::get_implementation`

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();

const implementation_type & get_implementation() const;
```

5.81.9.1 `basic_waitable_timer::get_implementation` (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.81.9.2 `basic_waitable_timer::get_implementation` (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.81.10 basic_waitable_timer::get_io_service

Inherited from basic_io_object.

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the **io_service** object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the **io_service** object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.81.11 basic_waitable_timer::get_service

Get the service associated with the I/O object.

```
service_type & get_service();

const service_type & get_service() const;
```

5.81.11.1 basic_waitable_timer::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.81.11.2 basic_waitable_timer::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.81.12 basic_waitable_timer::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.81.13 basic_waitable_timer::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/basic_waitable_timer.hpp

Convenience header: asio.hpp

5.81.14 basic_waitable_timer::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.81.15 basic_waitable_timer::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef WaitableTimerService service_type;
```

Requirements

Header: asio/basic_waitable_timer.hpp

Convenience header: asio.hpp

5.81.16 basic_waitable_timer::time_point

The time point type of the clock.

```
typedef clock_type::time_point time_point;
```

Requirements

Header: asio/basic_waitable_timer.hpp

Convenience header: asio.hpp

5.81.17 basic_waitable_timer::traits_type

The wait traits type.

```
typedef WaitTraits traits_type;
```

Requirements

Header: asio/basic_waitable_timer.hpp

Convenience header: asio.hpp

5.81.18 basic_waitable_timer::wait

Perform a blocking wait on the timer.

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

5.81.18.1 basic_waitable_timer::wait (1 of 2 overloads)

Perform a blocking wait on the timer.

```
void wait();
```

This function is used to wait for the timer to expire. This function blocks and does not return until the timer has expired.

Exceptions

asio::system_error Thrown on failure.

5.81.18.2 basic_waitable_timer::wait (2 of 2 overloads)

Perform a blocking wait on the timer.

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

This function is used to wait for the timer to expire. This function blocks and does not return until the timer has expired.

Parameters

ec Set to indicate what error occurred, if any.

5.82 basic_yield_context

Context object the represents the currently executing coroutine.

```
template<  
    typename Handler>  
class basic_yield_context
```

Types

Name	Description
callee_type	The coroutine callee type, used by the implementation.
caller_type	The coroutine caller type, used by the implementation.

Member Functions

Name	Description
<code>basic_yield_context</code>	Construct a yield context to represent the specified coroutine.
<code>operator[]</code>	Return a yield context that sets the specified error_code.

The `basic_yield_context` class is used to represent the currently executing stackful coroutine. A `basic_yield_context` may be passed as a handler to an asynchronous operation. For example:

```
template <typename Handler>
void my_coroutine(basic_yield_context<Handler> yield)
{
    ...
    std::size_t n = my_socket.async_read_some(buffer, yield);
    ...
}
```

The initiating function (`async_read_some` in the above example) suspends the current coroutine. The coroutine is resumed when the asynchronous operation completes, and the result of the operation is returned.

Requirements

Header: `asio/spawn.hpp`

Convenience header: None

5.82.1 `basic_yield_context::basic_yield_context`

Construct a yield context to represent the specified coroutine.

```
basic_yield_context(
    const detail::weak_ptr< callee_type > & coro,
    caller_type & ca,
    Handler & handler);
```

Most applications do not need to use this constructor. Instead, the `spawn()` function passes a yield context as an argument to the coroutine function.

5.82.2 `basic_yield_context::callee_type`

The coroutine callee type, used by the implementation.

```
typedef implementation_defined callee_type;
```

When using Boost.Coroutine v1, this type is:

```
typename coroutine<void()>
```

When using Boost.Coroutine v2 (unidirectional coroutines), this type is:

```
push_coroutine<void>
```

Requirements

Header: asio/spawn.hpp

Convenience header: None

5.82.3 basic_yield_context::caller_type

The coroutine caller type, used by the implementation.

```
typedef implementation_defined caller_type;
```

When using Boost.Coroutine v1, this type is:

```
typename coroutine<void()>::caller_type
```

When using Boost.Coroutine v2 (unidirectional coroutines), this type is:

```
pull_coroutine<void>
```

Requirements

Header: asio/spawn.hpp

Convenience header: None

5.82.4 basic_yield_context::operator[]

Return a yield context that sets the specified `error_code`.

```
basic_yield_context operator[](
    asio::error_code & ec) const;
```

By default, when a yield context is used with an asynchronous operation, a non-success `error_code` is converted to `system_error` and thrown. This operator may be used to specify an `error_code` object that should instead be set with the asynchronous operation's result. For example:

```
template <typename Handler>
void my_coroutine(basic_yield_context<Handler> yield)
{
    ...
    std::size_t n = my_socket.async_read_some(buffer, yield[ec]);
    if (ec)
    {
        // An error occurred.
    }
    ...
}
```

5.83 buffer

The `asio::buffer` function is used to create a `buffer` object to represent raw memory, an array of POD elements, a vector of POD elements, or a `std::string`.

```

mutable_buffers_1 buffer(
    const mutable_buffer & b);

mutable_buffers_1 buffer(
    const mutable_buffer & b,
    std::size_t max_size_in_bytes);

const_buffers_1 buffer(
    const const_buffer & b);

const_buffers_1 buffer(
    const const_buffer & b,
    std::size_t max_size_in_bytes);

mutable_buffers_1 buffer(
    void * data,
    std::size_t size_in_bytes);

const_buffers_1 buffer(
    const void * data,
    std::size_t size_in_bytes);

template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    PodType (&data) [N]);

template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    PodType (&data) [N],
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const PodType (&data) [N]);

template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const PodType (&data) [N],
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>

```

```

mutable_buffers_1 buffer(
    boost::array< PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    boost::array< PodType, N > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    boost::array< const PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    boost::array< const PodType, N > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const boost::array< PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const boost::array< PodType, N > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    std::array< PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    std::array< PodType, N > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    std::array< const PodType, N > & data);

```

```

template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    std::array< const PodType, N > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const std::array< PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const std::array< PodType, N > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    typename Allocator>
mutable_buffers_1 buffer(
    std::vector< PodType, Allocator > & data);

template<
    typename PodType,
    typename Allocator>
mutable_buffers_1 buffer(
    std::vector< PodType, Allocator > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    typename Allocator>
const_buffers_1 buffer(
    const std::vector< PodType, Allocator > & data);

template<
    typename PodType,
    typename Allocator>
const_buffers_1 buffer(
    const std::vector< PodType, Allocator > & data,
    std::size_t max_size_in_bytes);

template<
    typename Elem,
    typename Traits,
    typename Allocator>
const_buffers_1 buffer(
    const std::basic_string< Elem, Traits, Allocator > & data);

```

```

template<
    typename Elem,
    typename Traits,
    typename Allocator>
const_buffers_1 buffer(
    const std::basic_string< Elem, Traits, Allocator > & data,
    std::size_t max_size_in_bytes);

```

A buffer object represents a contiguous region of memory as a 2-tuple consisting of a pointer and size in bytes. A tuple of the form `{void*, size_t}` specifies a mutable (modifiable) region of memory. Similarly, a tuple of the form `{const void*, size_t}` specifies a const (non-modifiable) region of memory. These two forms correspond to the classes `mutable_buffer` and `const_buffer`, respectively. To mirror C++'s conversion rules, a `mutable_buffer` is implicitly convertible to a `const_buffer`, and the opposite conversion is not permitted.

The simplest use case involves reading or writing a single buffer of a specified size:

```
sock.send(asio::buffer(data, size));
```

In the above example, the return value of `asio::buffer` meets the requirements of the `ConstBufferSequence` concept so that it may be directly passed to the socket's write function. A buffer created for modifiable memory also meets the requirements of the `MutableBufferSequence` concept.

An individual buffer may be created from a builtin array, `std::vector`, `std::array` or `boost::array` of POD elements. This helps prevent buffer overruns by automatically determining the size of the buffer:

```

char d1[128];
size_t bytes_transferred = sock.receive(asio::buffer(d1));

std::vector<char> d2(128);
bytes_transferred = sock.receive(asio::buffer(d2));

std::array<char, 128> d3;
bytes_transferred = sock.receive(asio::buffer(d3));

boost::array<char, 128> d4;
bytes_transferred = sock.receive(asio::buffer(d4));

```

In all three cases above, the buffers created are exactly 128 bytes long. Note that a vector is *never* automatically resized when creating or using a buffer. The buffer size is determined using the vector's `size()` member function, and not its capacity.

Accessing Buffer Contents

The contents of a buffer may be accessed using the `buffer_size` and `buffer_cast` functions:

```

asio::mutable_buffer b1 = ...;
std::size_t s1 = asio::buffer_size(b1);
unsigned char* p1 = asio::buffer_cast<unsigned char*>(b1);

asio::const_buffer b2 = ...;
std::size_t s2 = asio::buffer_size(b2);
const void* p2 = asio::buffer_cast<const void*>(b2);

```

The `asio::buffer_cast` function permits violations of type safety, so uses of it in application code should be carefully considered.

For convenience, the `buffer_size` function also works on buffer sequences (that is, types meeting the `ConstBufferSequence` or `MutableBufferSequence` type requirements). In this case, the function returns the total size of all buffers in the sequence.

Buffer Copying

The `buffer_copy` function may be used to copy raw bytes between individual buffers and buffer sequences.

In particular, when used with the `buffer_size`, the `buffer_copy` function can be used to linearise a sequence of buffers. For example:

```
vector<const_buffer> buffers = ...;

vector<unsigned char> data(asio::buffer_size(buffers));
asio::buffer_copy(asio::buffer(data), buffers);
```

Note that `buffer_copy` is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

Buffer Invalidiation

A buffer object does not have any ownership of the memory it refers to. It is the responsibility of the application to ensure the memory region remains valid until it is no longer required for an I/O operation. When the memory is no longer available, the buffer is said to have been invalidated.

For the `asio::buffer` overloads that accept an argument of type `std::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 `asio::buffer` overloads that accept an argument of type `std::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).

Buffer Arithmetic

Buffer objects may be manipulated using simple arithmetic in a safe way which helps prevent buffer overruns. Consider an array initialised as follows:

```
boost::array<char, 6> a = { 'a', 'b', 'c', 'd', 'e' };
```

A buffer object `b1` created using:

```
b1 = asio::buffer(a);
```

represents the entire array, `{ 'a', 'b', 'c', 'd', 'e' }`. An optional second argument to the `asio::buffer` function may be used to limit the size, in bytes, of the buffer:

```
b2 = asio::buffer(a, 3);
```

such that `b2` represents the data `{ 'a', 'b', 'c' }`. Even if the size argument exceeds the actual size of the array, the size of the buffer object created will be limited to the array size.

An offset may be applied to an existing buffer to create a new one:

```
b3 = b1 + 2;
```

where `b3` will set to represent `{ 'c', 'd', 'e' }`. If the offset exceeds the size of the existing buffer, the newly created buffer will be empty.

Both an offset and size may be specified to create a buffer that corresponds to a specific range of bytes within an existing buffer:

```
b4 = asio::buffer(b1 + 1, 3);
```

so that `b4` will refer to the bytes `{ 'b', 'c', 'd' }`.

Buffers and Scatter-Gather I/O

To read or write using multiple buffers (i.e. scatter-gather I/O), multiple buffer objects may be assigned into a container that supports the `MutableBufferSequence` (for read) or `ConstBufferSequence` (for write) concepts:

```
char d1[128];
std::vector<char> d2(128);
boost::array<char, 128> d3;

boost::array<mutable_buffer, 3> bufs1 = {
   asio::buffer(d1),
   asio::buffer(d2),
   asio::buffer(d3) };
bytes_transferred = sock.receive(bufs1);

std::vector<const_buffer> bufs2;
bufs2.push_back(asio::buffer(d1));
bufs2.push_back(asio::buffer(d2));
bufs2.push_back(asio::buffer(d3));
bytes_transferred = sock.send(bufs2);
```

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.83.1 `buffer` (1 of 28 overloads)

Create a new modifiable buffer from an existing buffer.

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

Return Value

`mutable_buffers_1(b).`

5.83.2 `buffer` (2 of 28 overloads)

Create a new modifiable buffer from an existing buffer.

```
mutable_buffers_1 buffer(
    const mutable_buffer & b,
    std::size_t max_size_in_bytes);
```

Return Value

A `mutable_buffers_1` value equivalent to:

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

5.83.3 buffer (3 of 28 overloads)

Create a new non-modifiable buffer from an existing buffer.

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

Return Value

```
const_buffers_1(b).
```

5.83.4 buffer (4 of 28 overloads)

Create a new non-modifiable buffer from an existing buffer.

```
const_buffers_1 buffer(
    const const_buffer & b,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffers_1` value equivalent to:

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

5.83.5 buffer (5 of 28 overloads)

Create a new modifiable buffer that represents the given memory range.

```
mutable_buffers_1 buffer(
    void * data,
    std::size_t size_in_bytes);
```

Return Value

```
mutable_buffers_1(data, size_in_bytes).
```

5.83.6 buffer (6 of 28 overloads)

Create a new non-modifiable buffer that represents the given memory range.

```
const_buffers_1 buffer(
    const void * data,
    std::size_t size_in_bytes);
```

Return Value

```
const_buffers_1(data, size_in_bytes).
```

5.83.7 buffer (7 of 28 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    PodType (&data) [N]);
```

Return Value

A [mutable_buffers_1](#) value equivalent to:

```
mutable_buffers_1(
    static_cast<void*>(data),
    N * sizeof(PodType));
```

5.83.8 buffer (8 of 28 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    PodType (&data) [N],
    std::size_t max_size_in_bytes);
```

Return Value

A [mutable_buffers_1](#) value equivalent to:

```
mutable_buffers_1(
    static_cast<void*>(data),
    min(N * sizeof(PodType), max_size_in_bytes));
```

5.83.9 buffer (9 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const PodType (&data) [N]);
```

Return Value

A [const_buffers_1](#) value equivalent to:

```
const_buffers_1(
    static_cast<const void*>(data),
    N * sizeof(PodType));
```

5.83.10 buffer (10 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const PodType (&data) [N],
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffers_1` value equivalent to:

```
const_buffers_1(
    static_cast<const void*>(data),
    min(N * sizeof(PodType), max_size_in_bytes));
```

5.83.11 buffer (11 of 28 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    boost::array<PodType, N> & data);
```

Return Value

A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.12 buffer (12 of 28 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    boost::array<PodType, N> & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.13 buffer (13 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    boost::array< const PodType, N > & data);
```

Return Value

A `const_buffers_1` value equivalent to:

```
const_buffers_1(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.14 buffer (14 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    boost::array< const PodType, N > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffers_1` value equivalent to:

```
const_buffers_1(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.15 buffer (15 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const boost::array< PodType, N > & data);
```

Return Value

A `const_buffers_1` value equivalent to:

```
const_buffers_1(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.16 buffer (16 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const boost::array< PodType, N > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffers_1` value equivalent to:

```
const_buffers_1(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.17 buffer (17 of 28 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    std::array< PodType, N > & data);
```

Return Value

A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.18 buffer (18 of 28 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffers_1 buffer(
    std::array< PodType, N > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.19 buffer (19 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    std::array< const PodType, N > & data);
```

Return Value

A `const_buffers_1` value equivalent to:

```
const_buffers_1(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.20 buffer (20 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    std::array< const PodType, N > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffers_1` value equivalent to:

```
const_buffers_1(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.21 buffer (21 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const std::array< PodType, N > & data);
```

Return Value

A `const_buffers_1` value equivalent to:

```
const_buffers_1(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.22 buffer (22 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffers_1 buffer(
    const std::array<PodType, N> & data,
    std::size_t max_size_in_bytes);
```

Return Value

A [const_buffers_1](#) value equivalent to:

```
const_buffers_1(
    data.data(),
    std::min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.23 buffer (23 of 28 overloads)

Create a new modifiable buffer that represents the given POD vector.

```
template<
    typename PodType,
    typename Allocator>
mutable_buffers_1 buffer(
    std::vector<PodType, Allocator> & data);
```

Return Value

A [mutable_buffers_1](#) value equivalent to:

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

Remarks

The buffer is invalidated by any vector operation that would also invalidate iterators.

5.83.24 buffer (24 of 28 overloads)

Create a new modifiable buffer that represents the given POD vector.

```
template<
    typename PodType,
    typename Allocator>
mutable_buffers_1 buffer(
    std::vector<PodType, Allocator> & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `mutable_buffers_1` value equivalent to:

```
mutable_buffers_1(
    data.size() ? &data[0] : 0,
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

Remarks

The buffer is invalidated by any vector operation that would also invalidate iterators.

5.83.25 buffer (25 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD vector.

```
template<
    typename PodType,
    typename Allocator>
const_buffers_1 buffer(
    const std::vector< PodType, Allocator > & data);
```

Return Value

A `const_buffers_1` value equivalent to:

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

Remarks

The buffer is invalidated by any vector operation that would also invalidate iterators.

5.83.26 buffer (26 of 28 overloads)

Create a new non-modifiable buffer that represents the given POD vector.

```
template<
    typename PodType,
    typename Allocator>
const_buffers_1 buffer(
    const std::vector< PodType, Allocator > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffers_1` value equivalent to:

```
const_buffers_1(
    data.size() ? &data[0] : 0,
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

Remarks

The buffer is invalidated by any vector operation that would also invalidate iterators.

5.83.27 **buffer (27 of 28 overloads)**

Create a new non-modifiable buffer that represents the given string.

```
template<
    typename Elem,
    typename Traits,
    typename Allocator>
const_buffers_1 buffer(
    const std::basic_string< Elem, Traits, Allocator > & data);
```

Return Value

```
const_buffers_1(data.data(), data.size() * sizeof(Elem)).
```

Remarks

The buffer is invalidated by any non-const operation called on the given string object.

5.83.28 **buffer (28 of 28 overloads)**

Create a new non-modifiable buffer that represents the given string.

```
template<
    typename Elem,
    typename Traits,
    typename Allocator>
const_buffers_1 buffer(
    const std::basic_string< Elem, Traits, Allocator > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffers_1` value equivalent to:

```
const_buffers_1(
    data.data(),
    min(data.size() * sizeof(Elem), max_size_in_bytes));
```

Remarks

The buffer is invalidated by any non-const operation called on the given string object.

5.84 buffer_cast

The `asio::buffer_cast` function is used to obtain a pointer to the underlying memory region associated with a buffer.

```
template<
    typename PointerToPodType>
PointerToPodType buffer_cast(
    const mutable_buffer & b);

template<
    typename PointerToPodType>
PointerToPodType buffer_cast(
    const const_buffer & b);
```

Examples:

To access the memory of a non-modifiable buffer, use:

```
asio::const_buffer b1 = ...;
const unsigned char* p1 = asio::buffer_cast<const unsigned char*>(b1);
```

To access the memory of a modifiable buffer, use:

```
asio::mutable_buffer b2 = ...;
unsigned char* p2 = asio::buffer_cast<unsigned char*>(b2);
```

The `asio::buffer_cast` function permits violations of type safety, so uses of it in application code should be carefully considered.

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.84.1 buffer_cast (1 of 2 overloads)

Cast a non-modifiable buffer to a specified pointer to POD type.

```
template<
    typename PointerToPodType>
PointerToPodType buffer_cast(
    const mutable_buffer & b);
```

5.84.2 buffer_cast (2 of 2 overloads)

Cast a non-modifiable buffer to a specified pointer to POD type.

```
template<
    typename PointerToPodType>
PointerToPodType buffer_cast(
    const const_buffer & b);
```

5.85 buffer_copy

The `asio::buffer_copy` function is used to copy bytes from a source buffer (or buffer sequence) to a target buffer (or buffer sequence).

```
std::size_t buffer_copy(
    const mutable_buffer & target,
    const const_buffer & source);

std::size_t buffer_copy(
    const mutable_buffer & target,
    const const_buffers_1 & source);

std::size_t buffer_copy(
    const mutable_buffer & target,
    const mutable_buffer & source);

std::size_t buffer_copy(
    const mutable_buffer & target,
    const mutable_buffers_1 & source);

template<
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const mutable_buffer & target,
    const ConstBufferSequence & source);

std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const const_buffer & source);

std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const const_buffers_1 & source);

std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const mutable_buffer & source);

std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const mutable_buffers_1 & source);

template<
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const ConstBufferSequence & source);

template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
```

```
const MutableBufferSequence & target,
const const_buffer & source);

template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const const_buffers_1 & source);

template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const mutable_buffer & source);

template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const mutable_buffers_1 & source);

template<
    typename MutableBufferSequence,
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const ConstBufferSequence & source);

std::size_t buffer_copy(
    const mutable_buffer & target,
    const const_buffer & source,
    std::size_t max_bytes_to_copy);

std::size_t buffer_copy(
    const mutable_buffer & target,
    const const_buffers_1 & source,
    std::size_t max_bytes_to_copy);

std::size_t buffer_copy(
    const mutable_buffer & target,
    const mutable_buffer & source,
    std::size_t max_bytes_to_copy);

std::size_t buffer_copy(
    const mutable_buffer & target,
    const mutable_buffers_1 & source,
    std::size_t max_bytes_to_copy);

template<
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const mutable_buffer & target,
    const ConstBufferSequence & source,
```

```
std::size_t max_bytes_to_copy);

std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const const_buffer & source,
    std::size_t max_bytes_to_copy);

std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const mutable_buffer & source,
    std::size_t max_bytes_to_copy);

std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const mutable_buffers_1 & source,
    std::size_t max_bytes_to_copy);

template<
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const ConstBufferSequence & source,
    std::size_t max_bytes_to_copy);

template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const const_buffer & source,
    std::size_t max_bytes_to_copy);

template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const const_buffers_1 & source,
    std::size_t max_bytes_to_copy);

template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const mutable_buffer & source,
    std::size_t max_bytes_to_copy);

template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const mutable_buffers_1 & source,
    std::size_t max_bytes_to_copy);
```

```

const MutableBufferSequence & target,
const mutable_buffers_1 & source,
std::size_t max_bytes_to_copy);

template<
    typename MutableBufferSequence,
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const ConstBufferSequence & source,
    std::size_t max_bytes_to_copy);

```

The `buffer_copy` function is available in two forms:

- A 2-argument form: `buffer_copy(target, source)`
- A 3-argument form: `buffer_copy(target, source, max_bytes_to_copy)`

Both forms return the number of bytes actually copied. The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- If specified, `max_bytes_to_copy`.

This prevents buffer overflow, regardless of the buffer sizes used in the copy operation.

Note that `buffer_copy` is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.85.1 `buffer_copy` (1 of 30 overloads)

Copies bytes from a source buffer to a target buffer.

```

std::size_t buffer_copy(
    const mutable_buffer & target,
    const const_buffer & source);

```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.2 `buffer_copy` (2 of 30 overloads)

Copies bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffer & target,
    const const_buffers_1 & source);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.3 `buffer_copy` (3 of 30 overloads)

Copies bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffer & target,
    const mutable_buffer & source);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.4 `buffer_copy` (4 of 30 overloads)

Copies bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffer & target,
    const mutable_buffers_1 & source);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.5 `buffer_copy` (5 of 30 overloads)

Copies bytes from a source buffer sequence to a target buffer.

```
template<
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const mutable_buffer & target,
    const ConstBufferSequence & source);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer sequence representing the memory regions from which the bytes will be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.6 `buffer_copy` (6 of 30 overloads)

Copies bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const const_buffer & source);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.7 `buffer_copy` (7 of 30 overloads)

Copies bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const const_buffers_1 & source);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.8 `buffer_copy` (8 of 30 overloads)

Copies bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const mutable_buffer & source);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.9 `buffer_copy` (9 of 30 overloads)

Copies bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const mutable_buffers_1 & source);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.10 `buffer_copy` (10 of 30 overloads)

Copies bytes from a source buffer sequence to a target buffer.

```
template<
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const ConstBufferSequence & source);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer sequence representing the memory regions from which the bytes will be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.11 `buffer_copy` (11 of 30 overloads)

Copies bytes from a source buffer to a target buffer sequence.

```
template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const const_buffer & source);
```

Parameters

target A modifiable buffer sequence representing the memory regions to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.12 `buffer_copy` (12 of 30 overloads)

Copies bytes from a source buffer to a target buffer sequence.

```
template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const const_buffers_1 & source);
```

Parameters

target A modifiable buffer sequence representing the memory regions to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.13 `buffer_copy` (13 of 30 overloads)

Copies bytes from a source buffer to a target buffer sequence.

```
template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const mutable_buffer & source);
```

Parameters

target A modifiable buffer sequence representing the memory regions to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.14 `buffer_copy` (14 of 30 overloads)

Copies bytes from a source buffer to a target buffer sequence.

```
template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const mutable_buffers_1 & source);
```

Parameters

target A modifiable buffer sequence representing the memory regions to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.15 `buffer_copy` (15 of 30 overloads)

Copies bytes from a source buffer sequence to a target buffer sequence.

```
template<
    typename MutableBufferSequence,
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const ConstBufferSequence & source);
```

Parameters

target A modifiable buffer sequence representing the memory regions to which the bytes will be copied.

source A non-modifiable buffer sequence representing the memory regions from which the bytes will be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.16 `buffer_copy` (16 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffer & target,
    const const_buffer & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.17 `buffer_copy` (17 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffer & target,
    const const_buffers_1 & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.18 `buffer_copy` (18 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffer & target,
    const mutable_buffer & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.19 `buffer_copy` (19 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffer & target,
    const mutable_buffers_1 & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.20 `buffer_copy` (20 of 30 overloads)

Copies a limited number of bytes from a source buffer sequence to a target buffer.

```
template<
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const mutable_buffer & target,
    const ConstBufferSequence & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer sequence representing the memory regions from which the bytes will be copied.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.21 `buffer_copy` (21 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const const_buffer & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.22 `buffer_copy` (22 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const const_buffers_1 & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.23 `buffer_copy` (23 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const mutable_buffer & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.24 `buffer_copy` (24 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer.

```
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const mutable_buffers_1 & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.25 `buffer_copy` (25 of 30 overloads)

Copies a limited number of bytes from a source buffer sequence to a target buffer.

```
template<
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const mutable_buffers_1 & target,
    const ConstBufferSequence & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer representing the memory region to which the bytes will be copied.

source A non-modifiable buffer sequence representing the memory regions from which the bytes will be copied.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.26 `buffer_copy` (26 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer sequence.

```
template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const const_buffer & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer sequence representing the memory regions to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.27 `buffer_copy` (27 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer sequence.

```
template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const const_buffers_1 & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer sequence representing the memory regions to which the bytes will be copied.

source A non-modifiable buffer representing the memory region from which the bytes will be copied.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.28 `buffer_copy` (28 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer sequence.

```
template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const mutable_buffer & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer sequence representing the memory regions to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.29 `buffer_copy` (29 of 30 overloads)

Copies a limited number of bytes from a source buffer to a target buffer sequence.

```
template<
    typename MutableBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const mutable_buffers_1 & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer sequence representing the memory regions to which the bytes will be copied.

source A modifiable buffer representing the memory region from which the bytes will be copied. The contents of the source buffer will not be modified.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.85.30 `buffer_copy` (30 of 30 overloads)

Copies a limited number of bytes from a source buffer sequence to a target buffer sequence.

```
template<
    typename MutableBufferSequence,
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const ConstBufferSequence & source,
    std::size_t max_bytes_to_copy);
```

Parameters

target A modifiable buffer sequence representing the memory regions to which the bytes will be copied.

source A non-modifiable buffer sequence representing the memory regions from which the bytes will be copied.

max_bytes_to_copy The maximum number of bytes to be copied.

Return Value

The number of bytes copied.

Remarks

The number of bytes copied is the lesser of:

- `buffer_size(target)`
- `buffer_size(source)`
- `max_bytes_to_copy`

This function is implemented in terms of `memcpy`, and consequently it cannot be used to copy between overlapping memory regions.

5.86 `buffer_size`

The `asio::buffer_size` function determines the total number of bytes in a buffer or buffer sequence.

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

std::size_t buffer_size(
    const mutable_buffers_1 & b);

std::size_t buffer_size(
    const const_buffer & b);

std::size_t buffer_size(
    const const_buffers_1 & b);

template<
    typename BufferSequence>
std::size_t buffer_size(
    const BufferSequence & b);
```

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.86.1 `buffer_size (1 of 5 overloads)`

Get the number of bytes in a modifiable buffer.

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

5.86.2 buffer_size (2 of 5 overloads)

Get the number of bytes in a modifiable buffer.

```
std::size_t buffer_size(
    const mutable_buffers_1 & b);
```

5.86.3 buffer_size (3 of 5 overloads)

Get the number of bytes in a non-modifiable buffer.

```
std::size_t buffer_size(
    const const_buffer & b);
```

5.86.4 buffer_size (4 of 5 overloads)

Get the number of bytes in a non-modifiable buffer.

```
std::size_t buffer_size(
    const const_buffers_1 & b);
```

5.86.5 buffer_size (5 of 5 overloads)

Get the total number of bytes in a buffer sequence.

```
template<
    typename BufferSequence>
std::size_t buffer_size(
    const BufferSequence & b);
```

The BufferSequence template parameter may meet either of the ConstBufferSequence or MutableBufferSequence type requirements.

5.87 buffered_read_stream

Adds buffering to the read-related operations of a stream.

```
template<
    typename Stream>
class buffered_read_stream :
    noncopyable
```

Types

Name	Description
lowest_layer_type	The type of the lowest layer.
next_layer_type	The type of the next layer.

Member Functions

Name	Description
<code>async_fill</code>	Start an asynchronous fill.
<code>async_read_some</code>	Start an asynchronous read. The buffer into which the data will be read must be valid for the lifetime of the asynchronous operation.
<code>async_write_some</code>	Start an asynchronous write. The data being written must be valid for the lifetime of the asynchronous operation.
<code>buffered_read_stream</code>	Construct, passing the specified argument to initialise the next layer.
<code>close</code>	Close the stream.
<code>fill</code>	Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation. Throws an exception on failure. Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation, or 0 if an error occurred.
<code>get_io_service</code>	Get the io_service associated with the object.
<code>in_avail</code>	Determine the amount of data that may be read without blocking.
<code>lowest_layer</code>	Get a reference to the lowest layer. Get a const reference to the lowest layer.
<code>next_layer</code>	Get a reference to the next layer.
<code>peek</code>	Peek at the incoming data on the stream. Returns the number of bytes read. Throws an exception on failure. Peek at the incoming data on the stream. Returns the number of bytes read, or 0 if an error occurred.
<code>read_some</code>	Read some data from the stream. Returns the number of bytes read. Throws an exception on failure. Read some data from the stream. Returns the number of bytes read or 0 if an error occurred.
<code>write_some</code>	Write the given data to the stream. Returns the number of bytes written. Throws an exception on failure. Write the given data to the stream. Returns the number of bytes written, or 0 if an error occurred.

Data Members

Name	Description
default_buffer_size	The default buffer size.

The `buffered_read_stream` class template can be used to add buffering to the synchronous and asynchronous read operations of a stream.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/buffered_read_stream.hpp`

Convenience header: `asio.hpp`

5.87.1 buffered_read_stream::async_fill

Start an asynchronous fill.

```
template<
    typename ReadHandler>
void-or-deduced async_fill(
    ReadHandler handler);
```

5.87.2 buffered_read_stream::async_read_some

Start an asynchronous read. The buffer into which the data will be read must be valid for the lifetime of the asynchronous operation.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

5.87.3 buffered_read_stream::async_write_some

Start an asynchronous write. The data being written must be valid for the lifetime of the asynchronous operation.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some(
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

5.87.4 buffered_read_stream::buffered_read_stream

Construct, passing the specified argument to initialise the next layer.

```
template<
    typename Arg>
explicit buffered_read_stream(
    Arg & a);
```



```
template<
    typename Arg>
buffered_read_stream(
    Arg & a,
    std::size_t buffer_size);
```

5.87.4.1 buffered_read_stream::buffered_read_stream (1 of 2 overloads)

Construct, passing the specified argument to initialise the next layer.

```
template<
    typename Arg>
buffered_read_stream(
    Arg & a);
```

5.87.4.2 buffered_read_stream::buffered_read_stream (2 of 2 overloads)

Construct, passing the specified argument to initialise the next layer.

```
template<
    typename Arg>
buffered_read_stream(
    Arg & a,
    std::size_t buffer_size);
```

5.87.5 buffered_read_stream::close

Close the stream.

```
void close();
```



```
asio::error_code close(
    asio::error_code & ec);
```

5.87.5.1 buffered_read_stream::close (1 of 2 overloads)

Close the stream.

```
void close();
```

5.87.5.2 buffered_read_stream::close (2 of 2 overloads)

Close the stream.

```
asio::error_code close(
    asio::error_code & ec);
```

5.87.6 buffered_read_stream::default_buffer_size

The default buffer size.

```
static const std::size_t default_buffer_size = implementation_defined;
```

5.87.7 buffered_read_stream::fill

Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation. Throws an exception on failure.

```
std::size_t fill();
```

Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation, or 0 if an error occurred.

```
std::size_t fill(  
    asio::error_code & ec);
```

5.87.7.1 buffered_read_stream::fill (1 of 2 overloads)

Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation. Throws an exception on failure.

```
std::size_t fill();
```

5.87.7.2 buffered_read_stream::fill (2 of 2 overloads)

Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation, or 0 if an error occurred.

```
std::size_t fill(  
    asio::error_code & ec);
```

5.87.8 buffered_read_stream::get_io_service

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

5.87.9 buffered_read_stream::in_avail

Determine the amount of data that may be read without blocking.

```
std::size_t in_avail();
```

```
std::size_t in_avail(  
    asio::error_code & ec);
```

5.87.9.1 buffered_read_stream::in_avail (1 of 2 overloads)

Determine the amount of data that may be read without blocking.

```
std::size_t in_avail();
```

5.87.9.2 buffered_read_stream::in_avail (2 of 2 overloads)

Determine the amount of data that may be read without blocking.

```
std::size_t in_avail(  
    asio::error_code & ec);
```

5.87.10 buffered_read_stream::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.87.10.1 buffered_read_stream::lowest_layer (1 of 2 overloads)

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

5.87.10.2 buffered_read_stream::lowest_layer (2 of 2 overloads)

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.87.11 buffered_read_stream::lowest_layer_type

The type of the lowest layer.

```
typedef next_layer_type::lowest_layer_type lowest_layer_type;
```

Requirements

Header: asio/buffered_read_stream.hpp

Convenience header: asio.hpp

5.87.12 buffered_read_stream::next_layer

Get a reference to the next layer.

```
next_layer_type & next_layer();
```

5.87.13 buffered_read_stream::next_layer_type

The type of the next layer.

```
typedef remove_reference< Stream >::type next_layer_type;
```

Requirements

Header: asio/buffered_read_stream.hpp

Convenience header: asio.hpp

5.87.14 buffered_read_stream::peek

Peek at the incoming data on the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers);
```

Peek at the incoming data on the stream. Returns the number of bytes read, or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.87.14.1 buffered_read_stream::peek (1 of 2 overloads)

Peek at the incoming data on the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers);
```

5.87.14.2 buffered_read_stream::peek (2 of 2 overloads)

Peek at the incoming data on the stream. Returns the number of bytes read, or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.87.15 buffered_read_stream::read_some

Read some data from the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);
```

Read some data from the stream. Returns the number of bytes read or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.87.15.1 buffered_read_stream::read_some (1 of 2 overloads)

Read some data from the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);
```

5.87.15.2 buffered_read_stream::read_some (2 of 2 overloads)

Read some data from the stream. Returns the number of bytes read or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.87.16 buffered_read_stream::write_some

Write the given data to the stream. Returns the number of bytes written. Throws an exception on failure.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

Write the given data to the stream. Returns the number of bytes written, or 0 if an error occurred.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.87.16.1 buffered_read_stream::write_some (1 of 2 overloads)

Write the given data to the stream. Returns the number of bytes written. Throws an exception on failure.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

5.87.16.2 buffered_read_stream::write_some (2 of 2 overloads)

Write the given data to the stream. Returns the number of bytes written, or 0 if an error occurred.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.88 buffered_stream

Adds buffering to the read- and write-related operations of a stream.

```
template<
    typename Stream>
class buffered_stream :
    noncopyable
```

Types

Name	Description
lowest_layer_type	The type of the lowest layer.
next_layer_type	The type of the next layer.

Member Functions

Name	Description
async_fill	Start an asynchronous fill.
async_flush	Start an asynchronous flush.
async_read_some	Start an asynchronous read. The buffer into which the data will be read must be valid for the lifetime of the asynchronous operation.
async_write_some	Start an asynchronous write. The data being written must be valid for the lifetime of the asynchronous operation.
buffered_stream	Construct, passing the specified argument to initialise the next layer.
close	Close the stream.
fill	Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation. Throws an exception on failure. Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation, or 0 if an error occurred.

Name	Description
<code>flush</code>	Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation. Throws an exception on failure. Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation, or 0 if an error occurred.
<code>get_io_service</code>	Get the <code>io_service</code> associated with the object.
<code>in_avail</code>	Determine the amount of data that may be read without blocking.
<code>lowest_layer</code>	Get a reference to the lowest layer. Get a const reference to the lowest layer.
<code>next_layer</code>	Get a reference to the next layer.
<code>peek</code>	Peek at the incoming data on the stream. Returns the number of bytes read. Throws an exception on failure. Peek at the incoming data on the stream. Returns the number of bytes read, or 0 if an error occurred.
<code>read_some</code>	Read some data from the stream. Returns the number of bytes read. Throws an exception on failure. Read some data from the stream. Returns the number of bytes read or 0 if an error occurred.
<code>write_some</code>	Write the given data to the stream. Returns the number of bytes written. Throws an exception on failure. Write the given data to the stream. Returns the number of bytes written, or 0 if an error occurred.

The `buffered_stream` class template can be used to add buffering to the synchronous and asynchronous read and write operations of a stream.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/buffered_stream.hpp`

Convenience header: `asio.hpp`

5.88.1 `buffered_stream::async_fill`

Start an asynchronous fill.

```
template<
    typename ReadHandler>
void-or-deduced async_fill(
    ReadHandler handler);
```

5.88.2 buffered_stream::async_flush

Start an asynchronous flush.

```
template<
    typename WriteHandler>
void-or-deduced async_flush(
    WriteHandler handler);
```

5.88.3 buffered_stream::async_read_some

Start an asynchronous read. The buffer into which the data will be read must be valid for the lifetime of the asynchronous operation.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

5.88.4 buffered_stream::async_write_some

Start an asynchronous write. The data being written must be valid for the lifetime of the asynchronous operation.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some(
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

5.88.5 buffered_stream::buffered_stream

Construct, passing the specified argument to initialise the next layer.

```
template<
    typename Arg>
explicit buffered_stream(
    Arg & a);

template<
    typename Arg>
explicit buffered_stream(
    Arg & a,
    std::size_t read_buffer_size,
    std::size_t write_buffer_size);
```

5.88.5.1 buffered_stream::buffered_stream (1 of 2 overloads)

Construct, passing the specified argument to initialise the next layer.

```
template<
    typename Arg>
buffered_stream(
    Arg & a);
```

5.88.5.2 buffered_stream::buffered_stream (2 of 2 overloads)

Construct, passing the specified argument to initialise the next layer.

```
template<
    typename Arg>
buffered_stream(
    Arg & a,
    std::size_t read_buffer_size,
    std::size_t write_buffer_size);
```

5.88.6 buffered_stream::close

Close the stream.

```
void close();

asio::error_code close(
    asio::error_code & ec);
```

5.88.6.1 buffered_stream::close (1 of 2 overloads)

Close the stream.

```
void close();
```

5.88.6.2 buffered_stream::close (2 of 2 overloads)

Close the stream.

```
asio::error_code close(
    asio::error_code & ec);
```

5.88.7 buffered_stream::fill

Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation. Throws an exception on failure.

```
std::size_t fill();
```

Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation, or 0 if an error occurred.

```
std::size_t fill(
    asio::error_code & ec);
```

5.88.7.1 buffered_stream::fill (1 of 2 overloads)

Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation. Throws an exception on failure.

```
std::size_t fill();
```

5.88.7.2 buffered_stream::fill (2 of 2 overloads)

Fill the buffer with some data. Returns the number of bytes placed in the buffer as a result of the operation, or 0 if an error occurred.

```
std::size_t fill(  
    asio::error_code & ec);
```

5.88.8 buffered_stream::flush

Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation. Throws an exception on failure.

```
std::size_t flush();
```

Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation, or 0 if an error occurred.

```
std::size_t flush(  
    asio::error_code & ec);
```

5.88.8.1 buffered_stream::flush (1 of 2 overloads)

Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation. Throws an exception on failure.

```
std::size_t flush();
```

5.88.8.2 buffered_stream::flush (2 of 2 overloads)

Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation, or 0 if an error occurred.

```
std::size_t flush(  
    asio::error_code & ec);
```

5.88.9 buffered_stream::get_io_service

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

5.88.10 buffered_stream::in_avail

Determine the amount of data that may be read without blocking.

```
std::size_t in_avail();  
  
std::size_t in_avail(  
    asio::error_code & ec);
```

5.88.10.1 buffered_stream::in_avail (1 of 2 overloads)

Determine the amount of data that may be read without blocking.

```
std::size_t in_avail();
```

5.88.10.2 buffered_stream::in_avail (2 of 2 overloads)

Determine the amount of data that may be read without blocking.

```
std::size_t in_avail(  
    asio::error_code & ec);
```

5.88.11 buffered_stream::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.88.11.1 buffered_stream::lowest_layer (1 of 2 overloads)

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

5.88.11.2 buffered_stream::lowest_layer (2 of 2 overloads)

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.88.12 buffered_stream::lowest_layer_type

The type of the lowest layer.

```
typedef next_layer_type::lowest_layer_type lowest_layer_type;
```

Requirements

Header: asio/buffered_stream.hpp

Convenience header: asio.hpp

5.88.13 buffered_stream::next_layer

Get a reference to the next layer.

```
next_layer_type & next_layer();
```

5.88.14 buffered_stream::next_layer_type

The type of the next layer.

```
typedef remove_reference< Stream >::type next_layer_type;
```

Requirements

Header: asio/buffered_stream.hpp

Convenience header: asio.hpp

5.88.15 buffered_stream::peek

Peek at the incoming data on the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers);
```

Peek at the incoming data on the stream. Returns the number of bytes read, or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.88.15.1 buffered_stream::peek (1 of 2 overloads)

Peek at the incoming data on the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers);
```

5.88.15.2 buffered_stream::peek (2 of 2 overloads)

Peek at the incoming data on the stream. Returns the number of bytes read, or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.88.16 buffered_stream::read_some

Read some data from the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);
```

Read some data from the stream. Returns the number of bytes read or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.88.16.1 buffered_stream::read_some (1 of 2 overloads)

Read some data from the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);
```

5.88.16.2 buffered_stream::read_some (2 of 2 overloads)

Read some data from the stream. Returns the number of bytes read or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.88.17 buffered_stream::write_some

Write the given data to the stream. Returns the number of bytes written. Throws an exception on failure.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

Write the given data to the stream. Returns the number of bytes written, or 0 if an error occurred.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.88.17.1 buffered_stream::write_some (1 of 2 overloads)

Write the given data to the stream. Returns the number of bytes written. Throws an exception on failure.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

5.88.17.2 buffered_stream::write_some (2 of 2 overloads)

Write the given data to the stream. Returns the number of bytes written, or 0 if an error occurred.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.89 buffered_write_stream

Adds buffering to the write-related operations of a stream.

```
template<
    typename Stream>
class buffered_write_stream :
    noncopyable
```

Types

Name	Description
lowest_layer_type	The type of the lowest layer.
next_layer_type	The type of the next layer.

Member Functions

Name	Description
async_flush	Start an asynchronous flush.

Name	Description
async_read_some	Start an asynchronous read. The buffer into which the data will be read must be valid for the lifetime of the asynchronous operation.
async_write_some	Start an asynchronous write. The data being written must be valid for the lifetime of the asynchronous operation.
buffered_write_stream	Construct, passing the specified argument to initialise the next layer.
close	Close the stream.
flush	Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation. Throws an exception on failure. Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation, or 0 if an error occurred.
get_io_service	Get the io_service associated with the object.
in_avail	Determine the amount of data that may be read without blocking.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
next_layer	Get a reference to the next layer.
peek	Peek at the incoming data on the stream. Returns the number of bytes read. Throws an exception on failure. Peek at the incoming data on the stream. Returns the number of bytes read, or 0 if an error occurred.
read_some	Read some data from the stream. Returns the number of bytes read. Throws an exception on failure. Read some data from the stream. Returns the number of bytes read or 0 if an error occurred.
write_some	Write the given data to the stream. Returns the number of bytes written. Throws an exception on failure. Write the given data to the stream. Returns the number of bytes written, or 0 if an error occurred and the error handler did not throw.

Data Members

Name	Description
default_buffer_size	The default buffer size.

The `buffered_write_stream` class template can be used to add buffering to the synchronous and asynchronous write operations of a stream.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/buffered_write_stream.hpp

Convenience header: asio.hpp

5.89.1 buffered_write_stream::async_flush

Start an asynchronous flush.

```
template<
    typename WriteHandler>
void-or-deduced async_flush(
    WriteHandler handler);
```

5.89.2 buffered_write_stream::async_read_some

Start an asynchronous read. The buffer into which the data will be read must be valid for the lifetime of the asynchronous operation.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

5.89.3 buffered_write_stream::async_write_some

Start an asynchronous write. The data being written must be valid for the lifetime of the asynchronous operation.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some(
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

5.89.4 buffered_write_stream::buffered_write_stream

Construct, passing the specified argument to initialise the next layer.

```
template<
    typename Arg>
explicit buffered_write_stream(
    Arg & a);
```

```
template<
    typename Arg>
buffered_write_stream(
    Arg & a,
    std::size_t buffer_size);
```

5.89.4.1 buffered_write_stream::buffered_write_stream (1 of 2 overloads)

Construct, passing the specified argument to initialise the next layer.

```
template<
    typename Arg>
buffered_write_stream(
    Arg & a);
```

5.89.4.2 buffered_write_stream::buffered_write_stream (2 of 2 overloads)

Construct, passing the specified argument to initialise the next layer.

```
template<
    typename Arg>
buffered_write_stream(
    Arg & a,
    std::size_t buffer_size);
```

5.89.5 buffered_write_stream::close

Close the stream.

```
void close();

asio::error_code close(
    asio::error_code & ec);
```

5.89.5.1 buffered_write_stream::close (1 of 2 overloads)

Close the stream.

```
void close();
```

5.89.5.2 buffered_write_stream::close (2 of 2 overloads)

Close the stream.

```
asio::error_code close(
    asio::error_code & ec);
```

5.89.6 buffered_write_stream::default_buffer_size

The default buffer size.

```
static const std::size_t default_buffer_size = implementation_defined;
```

5.89.7 buffered_write_stream::flush

Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation. Throws an exception on failure.

```
std::size_t flush();
```

Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation, or 0 if an error occurred.

```
std::size_t flush(  
    asio::error_code & ec);
```

5.89.7.1 buffered_write_stream::flush (1 of 2 overloads)

Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation. Throws an exception on failure.

```
std::size_t flush();
```

5.89.7.2 buffered_write_stream::flush (2 of 2 overloads)

Flush all data from the buffer to the next layer. Returns the number of bytes written to the next layer on the last write operation, or 0 if an error occurred.

```
std::size_t flush(  
    asio::error_code & ec);
```

5.89.8 buffered_write_stream::get_io_service

Get the [io_service](#) associated with the object.

```
asio::io_service & get_io_service();
```

5.89.9 buffered_write_stream::in_avail

Determine the amount of data that may be read without blocking.

```
std::size_t in_avail();
```

```
std::size_t in_avail(  
    asio::error_code & ec);
```

5.89.9.1 buffered_write_stream::in_avail (1 of 2 overloads)

Determine the amount of data that may be read without blocking.

```
std::size_t in_avail();
```

5.89.9.2 buffered_write_stream::in_avail (2 of 2 overloads)

Determine the amount of data that may be read without blocking.

```
std::size_t in_avail(  
    asio::error_code & ec);
```

5.89.10 buffered_write_stream::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.89.10.1 buffered_write_stream::lowest_layer (1 of 2 overloads)

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

5.89.10.2 buffered_write_stream::lowest_layer (2 of 2 overloads)

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.89.11 buffered_write_stream::lowest_layer_type

The type of the lowest layer.

```
typedef next_layer_type::lowest_layer_type lowest_layer_type;
```

Requirements

Header: asio/buffered_write_stream.hpp

Convenience header: asio.hpp

5.89.12 buffered_write_stream::next_layer

Get a reference to the next layer.

```
next_layer_type & next_layer();
```

5.89.13 buffered_write_stream::next_layer_type

The type of the next layer.

```
typedef remove_reference< Stream >::type next_layer_type;
```

Requirements

Header: asio/buffered_write_stream.hpp

Convenience header: asio.hpp

5.89.14 buffered_write_stream::peek

Peek at the incoming data on the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers);
```

Peek at the incoming data on the stream. Returns the number of bytes read, or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.89.14.1 buffered_write_stream::peek (1 of 2 overloads)

Peek at the incoming data on the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers);
```

5.89.14.2 buffered_write_stream::peek (2 of 2 overloads)

Peek at the incoming data on the stream. Returns the number of bytes read, or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t peek(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.89.15 buffered_write_stream::read_some

Read some data from the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);
```

Read some data from the stream. Returns the number of bytes read or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.89.15.1 buffered_write_stream::read_some (1 of 2 overloads)

Read some data from the stream. Returns the number of bytes read. Throws an exception on failure.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);
```

5.89.15.2 buffered_write_stream::read_some (2 of 2 overloads)

Read some data from the stream. Returns the number of bytes read or 0 if an error occurred.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.89.16 buffered_write_stream::write_some

Write the given data to the stream. Returns the number of bytes written. Throws an exception on failure.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

Write the given data to the stream. Returns the number of bytes written, or 0 if an error occurred and the error handler did not throw.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.89.16.1 buffered_write_stream::write_some (1 of 2 overloads)

Write the given data to the stream. Returns the number of bytes written. Throws an exception on failure.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

5.89.16.2 buffered_write_stream::write_some (2 of 2 overloads)

Write the given data to the stream. Returns the number of bytes written, or 0 if an error occurred and the error handler did not throw.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.90 buffers_begin

Construct an iterator representing the beginning of the buffers' data.

```
template<
    typename BufferSequence>
buffers_iterator< BufferSequence > buffers_begin(
    const BufferSequence & buffers);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.91 buffers_end

Construct an iterator representing the end of the buffers' data.

```
template<
    typename BufferSequence>
buffers_iterator< BufferSequence > buffers_end(
    const BufferSequence & buffers);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92 buffers_iterator

A random access iterator over the bytes in a buffer sequence.

```
template<
    typename BufferSequence,
    typename ByteType = char>
class buffers_iterator
```

Types

Name	Description
difference_type	The type used for the distance between two iterators.
iterator_category	The iterator category.
pointer	The type of the result of applying operator->() to the iterator.
reference	The type of the result of applying operator*() to the iterator.
value_type	The type of the value pointed to by the iterator.

Member Functions

Name	Description
begin	Construct an iterator representing the beginning of the buffers' data.
buffers_iterator	Default constructor. Creates an iterator in an undefined state.
end	Construct an iterator representing the end of the buffers' data.
operator *	Dereference an iterator.
operator++	Increment operator (prefix). Increment operator (postfix).
operator+=	Addition operator.
operator--	Decrement operator (prefix). Decrement operator (postfix).
operator-=	Subtraction operator.
operator->	Dereference an iterator.
operator[]	Access an individual element.

Friends

Name	Description
operator!=	Test two iterators for inequality.
operator+	Addition operator.
operator-	Subtraction operator.
operator<	Compare two iterators.
operator<=	Compare two iterators.

Name	Description
operator==	Test two iterators for equality.
operator>	Compare two iterators.
operator>=	Compare two iterators.

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.1 buffers_iterator::begin

Construct an iterator representing the beginning of the buffers' data.

```
static buffers_iterator begin(
    const BufferSequence & buffers);
```

5.92.2 buffers_iterator::buffers_iterator

Default constructor. Creates an iterator in an undefined state.

```
buffers_iterator();
```

5.92.3 buffers_iterator::difference_type

The type used for the distance between two iterators.

```
typedef std::ptrdiff_t difference_type;
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.4 buffers_iterator::end

Construct an iterator representing the end of the buffers' data.

```
static buffers_iterator end(
    const BufferSequence & buffers);
```

5.92.5 buffers_iterator::iterator_category

The iterator category.

```
typedef std::random_access_iterator_tag iterator_category;
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.6 buffers_iterator::operator*

Dereference an iterator.

```
reference operator *() const;
```

5.92.7 buffers_iterator::operator!=

Test two iterators for inequality.

```
friend bool operator!=(
    const buffers_iterator & a,
    const buffers_iterator & b);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.8 buffers_iterator::operator+

Addition operator.

```
friend buffers_iterator operator+
    (const buffers_iterator & iter,
     std::ptrdiff_t difference);

friend buffers_iterator operator+
    (std::ptrdiff_t difference,
     const buffers_iterator & iter);
```

5.92.8.1 buffers_iterator::operator+ (1 of 2 overloads)

Addition operator.

```
friend buffers_iterator operator+
    (const buffers_iterator & iter,
     std::ptrdiff_t difference);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.8.2 buffers_iterator::operator+ (2 of 2 overloads)

Addition operator.

```
friend buffers_iterator operator+
    std::ptrdiff_t difference,
    const buffers_iterator & iter);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.9 buffers_iterator::operator++

Increment operator (prefix).

```
buffers_iterator & operator++();
```

Increment operator (postfix).

```
buffers_iterator operator++(
    int );
```

5.92.9.1 buffers_iterator::operator++ (1 of 2 overloads)

Increment operator (prefix).

```
buffers_iterator & operator++();
```

5.92.9.2 buffers_iterator::operator++ (2 of 2 overloads)

Increment operator (postfix).

```
buffers_iterator operator++(
    int );
```

5.92.10 buffers_iterator::operator+=

Addition operator.

```
buffers_iterator & operator+=(  
    std::ptrdiff_t difference);
```

5.92.11 buffers_iterator::operator-

Subtraction operator.

```
friend buffers_iterator operator-(
    const buffers_iterator & iter,
    std::ptrdiff_t difference);
```

```
friend std::ptrdiff_t operator-(
    const buffers_iterator & a,
    const buffers_iterator & b);
```

5.92.11.1 buffers_iterator::operator- (1 of 2 overloads)

Subtraction operator.

```
friend buffers_iterator operator-
    const buffers_iterator & iter,
    std::ptrdiff_t difference);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.11.2 buffers_iterator::operator- (2 of 2 overloads)

Subtraction operator.

```
friend std::ptrdiff_t operator-
    const buffers_iterator & a,
    const buffers_iterator & b);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.12 buffers_iterator::operator--

Decrement operator (prefix).

```
buffers_iterator & operator--();
```

Decrement operator (postfix).

```
buffers_iterator operator--(
    int );
```

5.92.12.1 buffers_iterator::operator-- (1 of 2 overloads)

Decrement operator (prefix).

```
buffers_iterator & operator--();
```

5.92.12.2 buffers_iterator::operator-- (2 of 2 overloads)

Decrement operator (postfix).

```
buffers_iterator operator--(
    int );
```

5.92.13 buffers_iterator::operator-=

Subtraction operator.

```
buffers_iterator & operator-=
    std::ptrdiff_t difference);
```

5.92.14 buffers_iterator::operator->

Dereference an iterator.

```
pointer operator->() const;
```

5.92.15 buffers_iterator::operator<

Compare two iterators.

```
friend bool operator<
    const buffers_iterator & a,
    const buffers_iterator & b);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.16 buffers_iterator::operator<=

Compare two iterators.

```
friend bool operator<=
    const buffers_iterator & a,
    const buffers_iterator & b);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.17 buffers_iterator::operator==

Test two iterators for equality.

```
friend bool operator==
    const buffers_iterator & a,
    const buffers_iterator & b);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.18 buffers_iterator::operator>

Compare two iterators.

```
friend bool operator>(
    const buffers_iterator & a,
    const buffers_iterator & b);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.19 buffers_iterator::operator>=

Compare two iterators.

```
friend bool operator>=(
    const buffers_iterator & a,
    const buffers_iterator & b);
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.20 buffers_iterator::operator[]

Access an individual element.

```
reference operator[](  
    std::ptrdiff_t difference) const;
```

5.92.21 buffers_iterator::pointer

The type of the result of applying `operator->()` to the iterator.

```
typedef const_or_non_const_ByteType * pointer;
```

If the buffer sequence stores buffer objects that are convertible to `mutable_buffer`, this is a pointer to a non-const `ByteType`. Otherwise, a pointer to a const `ByteType`.

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.22 buffers_iterator::reference

The type of the result of applying `operator*()` to the iterator.

```
typedef const_or_non_const_ByteType & reference;
```

If the buffer sequence stores buffer objects that are convertible to `mutable_buffer`, this is a reference to a non-const `ByteType`. Otherwise, a reference to a const `ByteType`.

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.92.23 buffers_iterator::value_type

The type of the value pointed to by the iterator.

```
typedef ByteType value_type;
```

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.93 connect

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin);
```



```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    asio::error_code & ec);
```



```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    Iterator end);
```



```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    Iterator end,
```

```

asio::error_code & ec);

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    ConnectCondition connect_condition);

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    ConnectCondition connect_condition,
    asio::error_code & ec);

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    Iterator end,
    ConnectCondition connect_condition);

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    Iterator end,
    ConnectCondition connect_condition,
    asio::error_code & ec);

```

Requirements

Header: asio/connect.hpp

Convenience header: asio.hpp

5.93.1 connect (1 of 8 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```

template<
    typename Protocol,
    typename SocketService,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin);

```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

s The socket to be connected. If the socket is already open, it will be closed.

begin An iterator pointing to the start of a sequence of endpoints.

Return Value

On success, an iterator denoting the successfully connected endpoint. Otherwise, the end iterator.

Exceptions

asio::system_error Thrown on failure. If the sequence is empty, the associated `error_code` is `asio::error::not_found`. Otherwise, contains the error from the last connection attempt.

Remarks

This overload assumes that a default constructed object of type `Iterator` represents the end of the sequence. This is a valid assumption for iterator types such as `asio::ip::tcp::resolver::iterator`.

Example

```

tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::socket s(io_service);
asio::connect(s, r.resolve(q));

```

5.93.2 connect (2 of 8 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```

template<
    typename Protocol,
    typename SocketService,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    asio::error_code & ec);

```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

- s** The socket to be connected. If the socket is already open, it will be closed.
- begin** An iterator pointing to the start of a sequence of endpoints.
- ec** Set to indicate what error occurred, if any. If the sequence is empty, set to `asio::error::not_found`. Otherwise, contains the error from the last connection attempt.

Return Value

On success, an iterator denoting the successfully connected endpoint. Otherwise, the end iterator.

Remarks

This overload assumes that a default constructed object of type `Iterator` represents the end of the sequence. This is a valid assumption for iterator types such as `asio::ip::tcp::resolver::iterator`.

Example

```
tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::socket s(io_service);
asio::error_code ec;
asio::connect(s, r.resolve(q), ec);
if (ec)
{
    // An error occurred.
}
```

5.93.3 connect (3 of 8 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator>
Iterator connect(
    basic_socket<Protocol, SocketService> & s,
    Iterator begin,
    Iterator end);
```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

- s** The socket to be connected. If the socket is already open, it will be closed.
- begin** An iterator pointing to the start of a sequence of endpoints.
- end** An iterator pointing to the end of a sequence of endpoints.

Return Value

On success, an iterator denoting the successfully connected endpoint. Otherwise, the end iterator.

Exceptions

asio::system_error Thrown on failure. If the sequence is empty, the associated `error_code` is `asio::error::not_found`. Otherwise, contains the error from the last connection attempt.

Example

```
tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::resolver::iterator i = r.resolve(q), end;
tcp::socket s(io_service);
asio::connect(s, i, end);
```

5.93.4 connect (4 of 8 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator>
Iterator connect(
    basic_socket<Protocol, SocketService> & s,
    Iterator begin,
    Iterator end,
    asio::error_code & ec);
```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

s The socket to be connected. If the socket is already open, it will be closed.

begin An iterator pointing to the start of a sequence of endpoints.

end An iterator pointing to the end of a sequence of endpoints.

ec Set to indicate what error occurred, if any. If the sequence is empty, set to `asio::error::not_found`. Otherwise, contains the error from the last connection attempt.

Return Value

On success, an iterator denoting the successfully connected endpoint. Otherwise, the end iterator.

Example

```
tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::resolver::iterator i = r.resolve(q), end;
tcp::socket s(io_service);
asio::error_code ec;
asio::connect(s, i, end, ec);
if (ec)
{
    // An error occurred.
}
```

5.93.5 connect (5 of 8 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    ConnectCondition connect_condition);
```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

s The socket to be connected. If the socket is already open, it will be closed.

begin An iterator pointing to the start of a sequence of endpoints.

connect_condition A function object that is called prior to each connection attempt. The signature of the function object must be:

```
Iterator connect_condition(
    const asio::error_code& ec,
    Iterator next);
```

The `ec` parameter contains the result from the most recent `connect` operation. Before the first connection attempt, `ec` is always set to indicate success. The `next` parameter is an iterator pointing to the next endpoint to be tried. The function object should return the next iterator, but is permitted to return a different iterator so that endpoints may be skipped. The implementation guarantees that the function object will never be called with the `end` iterator.

Return Value

On success, an iterator denoting the successfully connected endpoint. Otherwise, the `end` iterator.

Exceptions

asio::system_error Thrown on failure. If the sequence is empty, the associated `error_code` is `asio::error::not_found`. Otherwise, contains the error from the last connection attempt.

Remarks

This overload assumes that a default constructed object of type `Iterator` represents the end of the sequence. This is a valid assumption for iterator types such as `asio::ip::tcp::resolver::iterator`.

Example

The following connect condition function object can be used to output information about the individual connection attempts:

```
struct my_connect_condition
{
    template <typename Iterator>
    Iterator operator()(const asio::error_code& ec,
                         Iterator next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next->endpoint() << std::endl;
        return next;
    }
};
```

It would be used with the `asio::connect` function as follows:

```
tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::socket s(io_service);
tcp::resolver::iterator i = asio::connect(
    s, r.resolve(q), my_connect_condition());
std::cout << "Connected to: " << i->endpoint() << std::endl;
```

5.93.6 connect (6 of 8 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    ConnectCondition connect_condition,
    asio::error_code & ec);
```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

s The socket to be connected. If the socket is already open, it will be closed.

begin An iterator pointing to the start of a sequence of endpoints.

connect_condition A function object that is called prior to each connection attempt. The signature of the function object must be:

```
Iterator connect_condition(
    const asio::error_code& ec,
    Iterator next);
```

The `ec` parameter contains the result from the most recent connect operation. Before the first connection attempt, `ec` is always set to indicate success. The `next` parameter is an iterator pointing to the next endpoint to be tried. The function object should return the next iterator, but is permitted to return a different iterator so that endpoints may be skipped. The implementation guarantees that the function object will never be called with the end iterator.

- ec** Set to indicate what error occurred, if any. If the sequence is empty, set to `asio::error::not_found`. Otherwise, contains the error from the last connection attempt.

Return Value

On success, an iterator denoting the successfully connected endpoint. Otherwise, the end iterator.

Remarks

This overload assumes that a default constructed object of type `Iterator` represents the end of the sequence. This is a valid assumption for iterator types such as `asio::ip::tcp::resolver::iterator`.

Example

The following connect condition function object can be used to output information about the individual connection attempts:

```
struct my_connect_condition
{
    template <typename Iterator>
    Iterator operator()(const asio::error_code& ec, Iterator next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next->endpoint() << std::endl;
        return next;
    }
};
```

It would be used with the `asio::connect` function as follows:

```
tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::socket s(io_service);
asio::error_code ec;
tcp::resolver::iterator i = asio::connect(
    s, r.resolve(q), my_connect_condition(), ec);
if (ec)
{
    // An error occurred.
}
else
{
    std::cout << "Connected to: " << i->endpoint() << std::endl;
}
```

5.93.7 connect (7 of 8 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    Iterator end,
    ConnectCondition connect_condition);
```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

s The socket to be connected. If the socket is already open, it will be closed.

begin An iterator pointing to the start of a sequence of endpoints.

end An iterator pointing to the end of a sequence of endpoints.

connect_condition A function object that is called prior to each connection attempt. The signature of the function object must be:

```
Iterator connect_condition(
    const asio::error_code& ec,
    Iterator next);
```

The `ec` parameter contains the result from the most recent connect operation. Before the first connection attempt, `ec` is always set to indicate success. The `next` parameter is an iterator pointing to the next endpoint to be tried. The function object should return the next iterator, but is permitted to return a different iterator so that endpoints may be skipped. The implementation guarantees that the function object will never be called with the end iterator.

Return Value

On success, an iterator denoting the successfully connected endpoint. Otherwise, the end iterator.

Exceptions

asio::system_error Thrown on failure. If the sequence is empty, the associated `error_code` is `asio::error::not_found`. Otherwise, contains the error from the last connection attempt.

Example

The following connect condition function object can be used to output information about the individual connection attempts:

```
struct my_connect_condition
{
    template <typename Iterator>
    Iterator operator()(const asio::error_code& ec, Iterator next)
    {
```

```

    if (ec) std::cout << "Error: " << ec.message() << std::endl;
    std::cout << "Trying: " << next->endpoint() << std::endl;
    return next;
}
};

```

It would be used with the `asio::connect` function as follows:

```

tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::resolver::iterator i = r.resolve(q), end;
tcp::socket s(io_service);
i = asio::connect(s, i, end, my_connect_condition());
std::cout << "Connected to: " << i->endpoint() << std::endl;

```

5.93.8 connect (8 of 8 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```

template<
    typename Protocol,
    typename SocketService,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol, SocketService > & s,
    Iterator begin,
    Iterator end,
    ConnectCondition connect_condition,
    asio::error_code & ec);

```

This function attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the socket's `connect` member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

s The socket to be connected. If the socket is already open, it will be closed.

begin An iterator pointing to the start of a sequence of endpoints.

end An iterator pointing to the end of a sequence of endpoints.

connect_condition A function object that is called prior to each connection attempt. The signature of the function object must be:

```

Iterator connect_condition(
    const asio::error_code& ec,
    Iterator next);

```

The `ec` parameter contains the result from the most recent connect operation. Before the first connection attempt, `ec` is always set to indicate success. The `next` parameter is an iterator pointing to the next endpoint to be tried. The function object should return the next iterator, but is permitted to return a different iterator so that endpoints may be skipped. The implementation guarantees that the function object will never be called with the end iterator.

ec Set to indicate what error occurred, if any. If the sequence is empty, set to `asio::error::not_found`. Otherwise, contains the error from the last connection attempt.

Return Value

On success, an iterator denoting the successfully connected endpoint. Otherwise, the end iterator.

Example

The following connect condition function object can be used to output information about the individual connection attempts:

```
struct my_connect_condition
{
    template <typename Iterator>
    Iterator operator()(const asio::error_code& ec,
                         Iterator next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next->endpoint() << std::endl;
        return next;
    }
};
```

It would be used with the `asio::connect` function as follows:

```
tcp::resolver r(io_service);
tcp::resolver::query q("host", "service");
tcp::resolver::iterator i = r.resolve(q), end;
tcp::socket s(io_service);
asio::error_code ec;
i = asio::connect(s, i, end, my_connect_condition(), ec);
if (ec)
{
    // An error occurred.
}
else
{
    std::cout << "Connected to: " << i->endpoint() << std::endl;
}
```

5.94 const_buffer

Holds a buffer that cannot be modified.

```
class const_buffer
```

Member Functions

Name	Description
const_buffer	Construct an empty buffer. Construct a buffer to represent a given memory range. Construct a non-modifiable buffer from a modifiable one.

Related Functions

Name	Description
operator+	Create a new non-modifiable buffer that is offset from the start of another.

The `const_buffer` class provides a safe representation of a buffer that cannot be modified. It does not own the underlying data, and so is cheap to copy or assign.

Accessing Buffer Contents

The contents of a buffer may be accessed using the `buffer_size` and `buffer_cast` functions:

```
asio::const_buffer b1 = ...;
std::size_t s1 = asio::buffer_size(b1);
const unsigned char* p1 = asio::buffer_cast<const unsigned char*>(b1);
```

The `asio::buffer_cast` function permits violations of type safety, so uses of it in application code should be carefully considered.

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.94.1 `const_buffer::const_buffer`

Construct an empty buffer.

```
const_buffer();
```

Construct a buffer to represent a given memory range.

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

Construct a non-modifiable buffer from a modifiable one.

```
const_buffer(
    const mutable_buffer & b);
```

5.94.1.1 `const_buffer::const_buffer (1 of 3 overloads)`

Construct an empty buffer.

```
const_buffer();
```

5.94.1.2 `const_buffer::const_buffer (2 of 3 overloads)`

Construct a buffer to represent a given memory range.

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

5.94.1.3 `const_buffer::const_buffer (3 of 3 overloads)`

Construct a non-modifiable buffer from a modifiable one.

```
const_buffer(
    const mutable_buffer & b);
```

5.94.2 `const_buffer::operator+`

Create a new non-modifiable buffer that is offset from the start of another.

```
const_buffer operator+
    const const_buffer & b,
    std::size_t start);

const_buffer operator+
    std::size_t start,
    const const_buffer & b);
```

5.94.2.1 `const_buffer::operator+ (1 of 2 overloads)`

Create a new non-modifiable buffer that is offset from the start of another.

```
const_buffer operator+
    const const_buffer & b,
    std::size_t start);
```

5.94.2.2 `const_buffer::operator+ (2 of 2 overloads)`

Create a new non-modifiable buffer that is offset from the start of another.

```
const_buffer operator+
    std::size_t start,
    const const_buffer & b);
```

5.95 `const_buffers_1`

Adapts a single non-modifiable buffer so that it meets the requirements of the ConstBufferSequence concept.

```
class const_buffers_1 :
    public const_buffer
```

Types

Name	Description
<code>const_iterator</code>	A random-access iterator type that may be used to read elements.
<code>value_type</code>	The type for each element in the list of buffers.

Member Functions

Name	Description
<code>begin</code>	Get a random-access iterator to the first element.
<code>const_buffers_1</code>	Construct to represent a given memory range. Construct to represent a single non-modifiable buffer.
<code>end</code>	Get a random-access iterator for one past the last element.

Related Functions

Name	Description
<code>operator+</code>	Create a new non-modifiable buffer that is offset from the start of another.

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.95.1 `const_buffers_1::begin`

Get a random-access iterator to the first element.

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

5.95.2 `const_buffers_1::const_buffers_1`

Construct to represent a given memory range.

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

Construct to represent a single non-modifiable buffer.

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

5.95.2.1 `const_buffers_1::const_buffers_1 (1 of 2 overloads)`

Construct to represent a given memory range.

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

5.95.2.2 const_buffers_1::const_buffers_1 (2 of 2 overloads)

Construct to represent a single non-modifiable buffer.

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

5.95.3 const_buffers_1::const_iterator

A random-access iterator type that may be used to read elements.

```
typedef const const_buffer * const_iterator;
```

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.95.4 const_buffers_1::end

Get a random-access iterator for one past the last element.

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

5.95.5 const_buffers_1::operator+

Create a new non-modifiable buffer that is offset from the start of another.

```
const_buffer operator+
    const const_buffer & b,
    std::size_t start);

const_buffer operator+
    std::size_t start,
    const const_buffer & b);
```

5.95.5.1 const_buffers_1::operator+ (1 of 2 overloads)

Inherited from const_buffer.

Create a new non-modifiable buffer that is offset from the start of another.

```
const_buffer operator+
    const const_buffer & b,
    std::size_t start);
```

5.95.5.2 const_buffers_1::operator+ (2 of 2 overloads)

Inherited from const_buffer.

Create a new non-modifiable buffer that is offset from the start of another.

```
const_buffer operator+
    std::size_t start,
    const const_buffer & b);
```

5.95.6 const_buffers_1::value_type

The type for each element in the list of buffers.

```
typedef const_buffer value_type;
```

Member Functions

Name	Description
const_buffer	Construct an empty buffer. Construct a buffer to represent a given memory range. Construct a non-modifiable buffer from a modifiable one.

Related Functions

Name	Description
operator+	Create a new non-modifiable buffer that is offset from the start of another.

The `const_buffer` class provides a safe representation of a buffer that cannot be modified. It does not own the underlying data, and so is cheap to copy or assign.

Accessing Buffer Contents

The contents of a buffer may be accessed using the `buffer_size` and `buffer_cast` functions:

```
asio::const_buffer b1 = ...;
std::size_t s1 = asio::buffer_size(b1);
const unsigned char* p1 = asio::buffer_cast<const unsigned char*>(b1);
```

The `asio::buffer_cast` function permits violations of type safety, so uses of it in application code should be carefully considered.

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.96 coroutine

Provides support for implementing stackless coroutines.

```
class coroutine
```

Member Functions

Name	Description
coroutine	Constructs a coroutine in its initial state.
is_child	Returns true if the coroutine is the child of a fork.
is_complete	Returns true if the coroutine has reached its terminal state.
is_parent	Returns true if the coroutine is the parent of a fork.

The `coroutine` class may be used to implement stackless coroutines. The class itself is used to store the current state of the coroutine.

Coroutines are copy-constructible and assignable, and the space overhead is a single int. They can be used as a base class:

```
class session : coroutine
{
    ...
};
```

or as a data member:

```
class session
{
    ...
    coroutine coro_;
};
```

or even bound in as a function argument using lambdas or `bind()`. The important thing is that as the application maintains a copy of the object for as long as the coroutine must be kept alive.

Pseudo-keywords

A coroutine is used in conjunction with certain "pseudo-keywords", which are implemented as macros. These macros are defined by a header file:

```
#include <asio/yield.hpp>
```

and may conversely be undefined as follows:

```
#include <asio/unyield.hpp>
```

reenter

The `reenter` macro is used to define the body of a coroutine. It takes a single argument: a pointer or reference to a coroutine object. For example, if the base class is a coroutine object you may write:

```
reenter (this)
{
    ... coroutine body ...
}
```

and if a data member or other variable you can write:

```
reenter (coro_)
{
    ... coroutine body ...
}
```

When `reenter` is executed at runtime, control jumps to the location of the last `yield` or `fork`.

The coroutine body may also be a single statement, such as:

```
reenter (this) for (;;)
{
    ...
}
```

Limitation: The `reenter` macro is implemented using a switch. This means that you must take care when using local variables within the coroutine body. The local variable is not allowed in a position where reentering the coroutine could bypass the variable definition.

`yield statement`

This form of the `yield` keyword is often used with asynchronous operations:

```
yield socket_->async_read_some(buffer(*buffer_), *this);
```

This divides into four logical steps:

- `yield` saves the current state of the coroutine.
- The statement initiates the asynchronous operation.
- The resume point is defined immediately following the statement.
- Control is transferred to the end of the coroutine body.

When the asynchronous operation completes, the function object is invoked and `reenter` causes control to transfer to the resume point. It is important to remember to carry the coroutine state forward with the asynchronous operation. In the above snippet, the current class is a function object object with a coroutine object as base class or data member.

The statement may also be a compound statement, and this permits us to define local variables with limited scope:

```
yield
{
    mutable_buffers_1 b = buffer(*buffer_);
    socket_->async_read_some(b, *this);
}
```

`yield return expression ;`

This form of `yield` is often used in generators or coroutine-based parsers. For example, the function object:

```
struct interleave : coroutine
{
    istream& is1;
    istream& is2;
    char operator() (char c)
    {
        reenter (this) for (;;)
        {
            yield return is1.get();
            yield return is2.get();
        }
    }
};
```

defines a trivial coroutine that interleaves the characters from two input streams.

This type of `yield` divides into three logical steps:

- `yield` saves the current state of the coroutine.
- The resume point is defined immediately following the semicolon.
- The value of the expression is returned from the function.

yield ;

This form of `yield` is equivalent to the following steps:

- `yield` saves the current state of the coroutine.
- The resume point is defined immediately following the semicolon.
- Control is transferred to the end of the coroutine body.

This form might be applied when coroutines are used for cooperative threading and scheduling is explicitly managed. For example:

```
struct task : coroutine
{
    ...
    void operator()()
    {
        reenter(this)
        {
            while (... not finished ...)
            {
                ... do something ...
                yield;
                ... do some more ...
                yield;
            }
        }
    }
    ...
};

task t1, t2;
for(;;)
{
    t1();
    t2();
}
```

yield break ;

The final form of `yield` is used to explicitly terminate the coroutine. This form is comprised of two steps:

- `yield` sets the coroutine state to indicate termination.
- Control is transferred to the end of the coroutine body.

Once terminated, calls to `is_complete()` return true and the coroutine cannot be reentered.

Note that a coroutine may also be implicitly terminated if the coroutine body is exited without a `yield`, e.g. by `return`, `throw` or by running to the end of the body.

fork statement

The `fork` pseudo-keyword is used when "forking" a coroutine, i.e. splitting it into two (or more) copies. One use of `fork` is in a server, where a new coroutine is created to handle each client connection:

```

reenter (this)
{
    do
    {
        socket_.reset (new tcp::socket (io_service_));
        yield acceptor->async_accept (*socket_, *this);
        fork server(*this)();
    } while (is_parent());
    ... client-specific handling follows ...
}

```

The logical steps involved in a `fork` are:

- `fork` saves the current state of the coroutine.
- The statement creates a copy of the coroutine and either executes it immediately or schedules it for later execution.
- The resume point is defined immediately following the semicolon.
- For the "parent", control immediately continues from the next line.

The functions `is_parent()` and `is_child()` can be used to differentiate between parent and child. You would use these functions to alter subsequent control flow.

Note that `fork` doesn't do the actual forking by itself. It is the application's responsibility to create a clone of the coroutine and call it. The clone can be called immediately, as above, or scheduled for delayed execution using something like `io_service::post()`.

Alternate macro names

If preferred, an application can use macro names that follow a more typical naming convention, rather than the pseudo-keywords. These are:

- `ASIO_CORO_REENTER` instead of `reenter`
- `ASIO_CORO_YIELD` instead of `yield`
- `ASIO_CORO_FORK` instead of `fork`

Requirements

Header: `asio/coroutine.hpp`

Convenience header: `asio.hpp`

5.96.1 `coroutine::coroutine`

Constructs a coroutine in its initial state.

```
coroutine();
```

5.96.2 `coroutine::is_child`

Returns true if the coroutine is the child of a fork.

```
bool is_child() const;
```

5.96.3 coroutine::is_complete

Returns true if the coroutine has reached its terminal state.

```
bool is_complete() const;
```

5.96.4 coroutine::is_parent

Returns true if the coroutine is the parent of a fork.

```
bool is_parent() const;
```

5.97 datagram_socket_service

Default service implementation for a datagram socket.

```
template<
    typename Protocol>
class datagram_socket_service :
    public io_service::service
```

Types

Name	Description
endpoint_type	The endpoint type.
implementation_type	The type of a datagram socket.
native_handle_type	The native socket type.
native_type	(Deprecated: Use native_handle_type.) The native socket type.
protocol_type	The protocol type.

Member Functions

Name	Description
assign	Assign an existing native socket to a datagram socket.
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive.
async_receive_from	Start an asynchronous receive that will get the endpoint of the sender.
async_send	Start an asynchronous send.

Name	Description
async_send_to	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
bind	
cancel	Cancel all asynchronous operations associated with the socket.
close	Close a datagram socket implementation.
connect	Connect the datagram socket to the specified endpoint.
construct	Construct a new datagram socket implementation.
converting_move_construct	Move-construct a new datagram socket implementation from another protocol type.
datagram_socket_service	Construct a new datagram socket service for the specified io_service.
destroy	Destroy a datagram socket implementation.
get_io_service	Get the io_service object that owns the service.
get_option	Get a socket option.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint.
move_assign	Move-assign from another datagram socket implementation.
move_construct	Move-construct a new datagram socket implementation.
native	(Deprecated: Use native_handle().) Get the native socket implementation.
native_handle	Get the native socket implementation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	
receive	Receive some data from the peer.

Name	Description
receive_from	Receive a datagram with the endpoint of the sender.
remote_endpoint	Get the remote endpoint.
send	Send the given data to the peer.
send_to	Send a datagram to the specified endpoint.
set_option	Set a socket option.
shutdown	Disable sends or receives on the socket.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/datagram_socket_service.hpp

Convenience header: asio.hpp

5.97.1 datagram_socket_service::assign

Assign an existing native socket to a datagram socket.

```
asio::error_code assign(
    implementation_type & impl,
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.97.2 datagram_socket_service::async_connect

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
void-or-deduced async_connect(
    implementation_type & impl,
    const endpoint_type & peer_endpoint,
    ConnectHandler handler);
```

5.97.3 datagram_socket_service::async_receive

Start an asynchronous receive.

```

template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    ReadHandler handler);

```

5.97.4 datagram_socket_service::async_receive_from

Start an asynchronous receive that will get the endpoint of the sender.

```

template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive_from(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    ReadHandler handler);

```

5.97.5 datagram_socket_service::async_send

Start an asynchronous send.

```

template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);

```

5.97.6 datagram_socket_service::async_send_to

Start an asynchronous send.

```

template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send_to(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    WriteHandler handler);

```

5.97.7 datagram_socket_service::at_mark

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.97.8 `datagram_socket_service::available`

Determine the number of bytes available for reading.

```
std::size_t available(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.97.9 `datagram_socket_service::bind`

```
asio::error_code bind(
    implementation_type & impl,
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.97.10 `datagram_socket_service::cancel`

Cancel all asynchronous operations associated with the socket.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.97.11 `datagram_socket_service::close`

Close a datagram socket implementation.

```
asio::error_code close(
    implementation_type & impl,
    asio::error_code & ec);
```

5.97.12 `datagram_socket_service::connect`

Connect the datagram socket to the specified endpoint.

```
asio::error_code connect(
    implementation_type & impl,
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.97.13 `datagram_socket_service::construct`

Construct a new datagram socket implementation.

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

5.97.14 `datagram_socket_service::converting_move_construct`

Move-construct a new datagram socket implementation from another protocol type.

```
template<
    typename Protocol1>
void converting_move_construct(
    implementation_type & impl,
    typename datagram_socket_service< Protocol1 >::implementation_type & other_impl,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.97.15 `datagram_socket_service::datagram_socket_service`

Construct a new datagram socket service for the specified `io_service`.

```
datagram_socket_service(
    asio::io_service & io_service);
```

5.97.16 `datagram_socket_service::destroy`

Destroy a datagram socket implementation.

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

5.97.17 `datagram_socket_service::endpoint_type`

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/datagram_socket_service.hpp

Convenience header: asio.hpp

5.97.18 `datagram_socket_service::get_io_service`

Inherited from io_service.

Get the `io_service` object that owns the service.

```
asio::io_service & get_io_service();
```

5.97.19 `datagram_socket_service::get_option`

Get a socket option.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    const implementation_type & impl,
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.97.20 `datagram_socket_service::id`

The unique service identifier.

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

5.97.21 `datagram_socket_service::implementation_type`

The type of a datagram socket.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: `asio/datagram_socket_service.hpp`

Convenience header: `asio.hpp`

5.97.22 `datagram_socket_service::io_control`

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    implementation_type & impl,
    IoControlCommand & command,
    asio::error_code & ec);
```

5.97.23 `datagram_socket_service::is_open`

Determine whether the socket is open.

```
bool is_open(
    const implementation_type & impl) const;
```

5.97.24 `datagram_socket_service::local_endpoint`

Get the local endpoint.

```
endpoint_type local_endpoint(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.97.25 `datagram_socket_service::move_assign`

Move-assign from another datagram socket implementation.

```
void move_assign(
    implementation_type & impl,
    datagram_socket_service & other_service,
    implementation_type & other_impl);
```

5.97.26 datagram_socket_service::move_construct

Move-construct a new datagram socket implementation.

```
void move_construct(
    implementation_type & impl,
    implementation_type & other_impl);
```

5.97.27 datagram_socket_service::native

(Deprecated: Use native_handle().) Get the native socket implementation.

```
native_type native(
    implementation_type & impl);
```

5.97.28 datagram_socket_service::native_handle

Get the native socket implementation.

```
native_handle_type native_handle(
    implementation_type & impl);
```

5.97.29 datagram_socket_service::native_handle_type

The native socket type.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/datagram_socket_service.hpp

Convenience header: asio.hpp

5.97.30 datagram_socket_service::native_non_blocking

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the native socket implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.97.30.1 datagram_socket_service::native_non_blocking (1 of 2 overloads)

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

5.97.30.2 `datagram_socket_service::native_non_blocking` (2 of 2 overloads)

Sets the non-blocking mode of the native socket implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.97.31 `datagram_socket_service::native_type`

(Deprecated: Use `native_handle_type`.) The native socket type.

```
typedef implementation_defined native_type;
```

Requirements

Header: `asio/datagram_socket_service.hpp`

Convenience header: `asio.hpp`

5.97.32 `datagram_socket_service::non_blocking`

Gets the non-blocking mode of the socket.

```
bool non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.97.32.1 `datagram_socket_service::non_blocking` (1 of 2 overloads)

Gets the non-blocking mode of the socket.

```
bool non_blocking(
    const implementation_type & impl) const;
```

5.97.32.2 `datagram_socket_service::non_blocking` (2 of 2 overloads)

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.97.33 datagram_socket_service::open

```
asio::error_code open(
    implementation_type & impl,
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.97.34 datagram_socket_service::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/datagram_socket_service.hpp

Convenience header: asio.hpp

5.97.35 datagram_socket_service::receive

Receive some data from the peer.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.97.36 datagram_socket_service::receive_from

Receive a datagram with the endpoint of the sender.

```
template<
    typename MutableBufferSequence>
std::size_t receive_from(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.97.37 datagram_socket_service::remote_endpoint

Get the remote endpoint.

```
endpoint_type remote_endpoint(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.97.38 datagram_socket_service::send

Send the given data to the peer.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.97.39 datagram_socket_service::send_to

Send a datagram to the specified endpoint.

```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.97.40 datagram_socket_service::set_option

Set a socket option.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    implementation_type & impl,
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.97.41 datagram_socket_service::shutdown

Disable sends or receives on the socket.

```
asio::error_code shutdown(
    implementation_type & impl,
    socket_base::shutdown_type what,
    asio::error_code & ec);
```

5.98 deadline_timer

TypeDef for the typical usage of timer. Uses a UTC clock.

```
typedef basic_deadline_timer< boost::posix_time::ptime > deadline_timer;
```

Types

Name	Description
duration_type	The duration type.
implementation_type	The underlying implementation type of I/O object.
service_type	The type of the service that will be used to provide I/O operations.
time_type	The time type.
traits_type	The time traits type.

Member Functions

Name	Description
async_wait	Start an asynchronous wait on the timer.
basic_deadline_timer	Constructor. Constructor to set a particular expiry time as an absolute time. Constructor to set a particular expiry time relative to now.
cancel	Cancel any asynchronous operations that are waiting on the timer.
cancel_one	Cancels one asynchronous operation that is waiting on the timer.
expires_at	Get the timer's expiry time as an absolute time. Set the timer's expiry time as an absolute time.
expires_from_now	Get the timer's expiry time relative to now. Set the timer's expiry time relative to now.
get_io_service	Get the io_service associated with the object.
wait	Perform a blocking wait on the timer.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_deadline_timer` class template provides the ability to perform a blocking or asynchronous wait for a timer to expire.

A deadline timer is always in one of two states: "expired" or "not expired". If the `wait()` or `async_wait()` function is called on an expired timer, the wait operation will complete immediately.

Most applications will use the `deadline_timer` typedef.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Examples

Performing a blocking wait:

```
// Construct a timer without setting an expiry time.
asio::deadline_timer timer(io_service);

// Set an expiry time relative to now.
timer.expires_from_now(boost::posix_time::seconds(5));

// Wait for the timer to expire.
timer.wait();
```

Performing an asynchronous wait:

```
void handler(const asio::error_code& error)
{
    if (!error)
    {
        // Timer expired.
    }
}

...

// Construct a timer with an absolute expiry time.
asio::deadline_timer timer(io_service,
    boost::posix_time::time_from_string("2005-12-07 23:59:59.000"));

// Start an asynchronous wait.
timer.async_wait(handler);
```

Changing an active `deadline_timer`'s expiry time

Changing the expiry time of a timer while there are pending asynchronous waits causes those wait operations to be cancelled. To ensure that the action associated with the timer is performed only once, use something like this: used:

```

void on_some_event()
{
    if (my_timer.expires_from_now(seconds(5)) > 0)
    {
        // We managed to cancel the timer. Start new asynchronous wait.
        my_timer.async_wait(on_timeout);
    }
    else
    {
        // Too late, timer has already expired!
    }
}

void on_timeout(const asio::error_code& e)
{
    if (e != asio::error::operation_aborted)
    {
        // Timer was not cancelled, take necessary action.
    }
}

```

- The `asio::basic_deadline_timer::expires_from_now()` function cancels any pending asynchronous waits, and returns the number of asynchronous waits that were cancelled. If it returns 0 then you were too late and the wait handler has already been executed, or will soon be executed. If it returns 1 then the wait handler was successfully cancelled.
- If a wait handler is cancelled, the `error_code` passed to it contains the value `asio::error::operation_aborted`.

Requirements

Header: `asio/deadline_timer.hpp`

Convenience header: `asio.hpp`

5.99 deadline_timer_service

Default service implementation for a timer.

```

template<
    typename TimeType,
    typename TimeTraits = asio::time_traits<TimeType>>
class deadline_timer_service :
    public io_service::service

```

Types

Name	Description
<code>duration_type</code>	The duration type.
<code>implementation_type</code>	The implementation type of the deadline timer.
<code>time_type</code>	The time type.
<code>traits_type</code>	The time traits type.

Member Functions

Name	Description
<code>async_wait</code>	
<code>cancel</code>	Cancel any asynchronous wait operations associated with the timer.
<code>cancel_one</code>	Cancels one asynchronous wait operation associated with the timer.
<code>construct</code>	Construct a new timer implementation.
<code>deadline_timer_service</code>	Construct a new timer service for the specified <code>io_service</code> .
<code>destroy</code>	Destroy a timer implementation.
<code>expires_at</code>	Get the expiry time for the timer as an absolute time. Set the expiry time for the timer as an absolute time.
<code>expires_from_now</code>	Get the expiry time for the timer relative to now. Set the expiry time for the timer relative to now.
<code>get_io_service</code>	Get the <code>io_service</code> object that owns the service.
<code>wait</code>	

Data Members

Name	Description
<code>id</code>	The unique service identifier.

Requirements

Header: `asio/deadline_timer_service.hpp`

Convenience header: `asio.hpp`

5.99.1 `deadline_timer_service::async_wait`

```
template<
    typename WaitHandler>
void-or-deduced async_wait(  

    implementation_type & impl,  

    WaitHandler handler);
```

5.99.2 `deadline_timer_service::cancel`

Cancel any asynchronous wait operations associated with the timer.

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

5.99.3 deadline_timer_service::cancel_one

Cancels one asynchronous wait operation associated with the timer.

```
std::size_t cancel_one(
    implementation_type & impl,
    asio::error_code & ec);
```

5.99.4 deadline_timer_service::construct

Construct a new timer implementation.

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

5.99.5 deadline_timer_service::deadline_timer_service

Construct a new timer service for the specified [io_service](#).

```
deadline_timer_service(
    asio::io_service & io_service);
```

5.99.6 deadline_timer_service::destroy

Destroy a timer implementation.

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

5.99.7 deadline_timer_service::duration_type

The duration type.

```
typedef traits_type::duration_type duration_type;
```

Requirements

Header: asio/deadline_timer_service.hpp

Convenience header: asio.hpp

5.99.8 deadline_timer_service::expires_at

Get the expiry time for the timer as an absolute time.

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

Set the expiry time for the timer as an absolute time.

```
std::size_t expires_at(
    implementation_type & impl,
    const time_type & expiry_time,
    asio::error_code & ec);
```

5.99.8.1 deadline_timer_service::expires_at (1 of 2 overloads)

Get the expiry time for the timer as an absolute time.

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

5.99.8.2 deadline_timer_service::expires_at (2 of 2 overloads)

Set the expiry time for the timer as an absolute time.

```
std::size_t expires_at(
    implementation_type & impl,
    const time_type & expiry_time,
    asio::error_code & ec);
```

5.99.9 deadline_timer_service::expires_from_now

Get the expiry time for the timer relative to now.

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

Set the expiry time for the timer relative to now.

```
std::size_t expires_from_now(
    implementation_type & impl,
    const duration_type & expiry_time,
    asio::error_code & ec);
```

5.99.9.1 deadline_timer_service::expires_from_now (1 of 2 overloads)

Get the expiry time for the timer relative to now.

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

5.99.9.2 deadline_timer_service::expires_from_now (2 of 2 overloads)

Set the expiry time for the timer relative to now.

```
std::size_t expires_from_now(
    implementation_type & impl,
    const duration_type & expiry_time,
    asio::error_code & ec);
```

5.99.10 deadline_timer_service::get_io_service

Inherited from io_service.

Get the **io_service** object that owns the service.

```
asio::io_service & get_io_service();
```

5.99.11 deadline_timer_service::id

The unique service identifier.

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

5.99.12 deadline_timer_service::implementation_type

The implementation type of the deadline timer.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asio/deadline_timer_service.hpp

Convenience header: asio.hpp

5.99.13 deadline_timer_service::time_type

The time type.

```
typedef traits_type::time_type time_type;
```

Requirements

Header: asio/deadline_timer_service.hpp

Convenience header: asio.hpp

5.99.14 deadline_timer_service::traits_type

The time traits type.

```
typedef TimeTraits traits_type;
```

Requirements

Header: asio/deadline_timer_service.hpp

Convenience header: asio.hpp

5.99.15 deadline_timer_service::wait

```
void wait(
    implementation_type & impl,
    asio::error_code & ec);
```

5.100 error::addrinfo_category

```
static const asio::error_category & addrinfo_category = asio::error::get_addrinfo_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.101 error::addrinfo_errors

```
enum addrinfo_errors
```

Values

service_not_found The service is not supported for the given socket type.

socket_type_not_supported The socket type is not supported.

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.102 error::basic_errors

```
enum basic_errors
```

Values

access_denied Permission denied.

address_family_not_supported Address family not supported by protocol.

address_in_use Address already in use.

already_connected Transport endpoint is already connected.

already_started Operation already in progress.

broken_pipe Broken pipe.

connection_aborted A connection has been aborted.

connection_refused Connection refused.

connection_reset Connection reset by peer.

bad_descriptor Bad file descriptor.

fault Bad address.

host_unreachable No route to host.

in_progress Operation now in progress.

interrupted Interrupted system call.

invalid_argument Invalid argument.

message_size Message too long.

name_too_long The name was too long.

network_down Network is down.

network_reset Network dropped connection on reset.

network_unreachable Network is unreachable.

no_descriptors Too many open files.

no_buffer_space No buffer space available.

no_memory Cannot allocate memory.

no_permission Operation not permitted.

no_protocol_option Protocol not available.

no_such_device No such device.

not_connected Transport endpoint is not connected.

not_socket Socket operation on non-socket.

operation_aborted Operation cancelled.

operation_not_supported Operation not supported.

shut_down Cannot send after transport endpoint shutdown.

timed_out Connection timed out.

try_again Resource temporarily unavailable.

would_block The socket is marked non-blocking and the requested operation would block.

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.103 error::get_addrinfo_category

```
const asio::error_category & get_addrinfo_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.104 error::get_misc_category

```
const asio::error_category & get_misc_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.105 error::get_netdb_category

```
const asio::error_category & get_netdb_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.106 error::get_ssl_category

```
const asio::error_category & get_ssl_category();
```

Requirements

Header: asio/ssl/error.hpp

Convenience header: asio/ssl.hpp

5.107 error::get_system_category

```
const asio::error_category & get_system_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.108 error::make_error_code

```
asio::error_code make_error_code(
    basic_errors e);

asio::error_code make_error_code(
    netdb_errors e);

asio::error_code make_error_code(
    addrinfo_errors e);

asio::error_code make_error_code(
    misc_errors e);

asio::error_code make_error_code(
    ssl_errors e);
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.108.1 error::make_error_code (1 of 5 overloads)

```
asio::error_code make_error_code(
    basic_errors e);
```

5.108.2 error::make_error_code (2 of 5 overloads)

```
asio::error_code make_error_code(
    netdb_errors e);
```

5.108.3 error::make_error_code (3 of 5 overloads)

```
asio::error_code make_error_code(
    addrinfo_errors e);
```

5.108.4 error::make_error_code (4 of 5 overloads)

```
asio::error_code make_error_code(
    misc_errors e);
```

5.108.5 `error::make_error_code` (5 of 5 overloads)

```
asio::error_code make_error_code(  
    ssl_errors e);
```

5.109 `error::misc_category`

```
static const asio::error_category & misc_category = asio::error::get_misc_category();
```

Requirements

Header: `asio/error.hpp`

Convenience header: `asio.hpp`

5.110 `error::misc_errors`

```
enum misc_errors
```

Values

already_open Already open.

eof End of file or stream.

not_found Element not found.

fd_set_failure The descriptor cannot fit into the select system call's `fd_set`.

Requirements

Header: `asio/error.hpp`

Convenience header: `asio.hpp`

5.111 `error::netdb_category`

```
static const asio::error_category & netdb_category = asio::error::get_netdb_category();
```

Requirements

Header: `asio/error.hpp`

Convenience header: `asio.hpp`

5.112 `error::netdb_errors`

```
enum netdb_errors
```

Values

host_not_found Host not found (authoritative).

host_not_found_try_again Host not found (non-authoritative).

no_data The query is valid but does not have associated address data.

no_recovery A non-recoverable error occurred.

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.113 error::ssl_category

```
static const asio::error_category & ssl_category = asio::error::get_ssl_category();
```

Requirements

Header: asio/ssl/error.hpp

Convenience header: asio/ssl.hpp

5.114 error::ssl_errors

```
enum ssl_errors
```

Requirements

Header: asio/ssl/error.hpp

Convenience header: asio/ssl.hpp

5.115 error::system_category

```
static const asio::error_category & system_category = asio::error::get_system_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.116 error_category

Base class for all error categories.

```
class error_category :  
    noncopyable
```

Member Functions

Name	Description
message	Returns a string describing the error denoted by value.
name	Returns a string naming the error category.
operator!=	Inequality operator to compare two error categories.
operator==	Equality operator to compare two error categories.
~error_category	Destructor.

Requirements

Header: asio/error_code.hpp

Convenience header: asio.hpp

5.116.1 error_category::message

Returns a string describing the error denoted by value.

```
std::string message(
    int value) const;
```

5.116.2 error_category::name

Returns a string naming the error category.

```
const char * name() const;
```

5.116.3 error_category::operator!=

Inequality operator to compare two error categories.

```
bool operator!=(
    const error_category & rhs) const;
```

5.116.4 error_category::operator==

Equality operator to compare two error categories.

```
bool operator==( 
    const error_category & rhs) const;
```

5.116.5 error_category::~error_category

Destructor.

```
virtual ~error_category();
```

5.117 error_code

Class to represent an error code value.

```
class error_code
```

Types

Name	Description
unspecified_bool_type_t	
unspecified_bool_type	

Member Functions

Name	Description
category	Get the error category.
error_code	Default constructor. Construct with specific error code and category. Construct from an error code enum.
message	Get the message associated with the error.
operator unspecified_bool_type	Operator returns non-null if there is a non-success error code.
operator!	Operator to test if the error represents success.
unspecified_bool_true	
value	Get the error value.

Friends

Name	Description
operator!=	Inequality operator to compare two error objects.
operator==	Equality operator to compare two error objects.

Requirements

Header: asio/error_code.hpp

Convenience header: asio.hpp

5.117.1 error_code::category

Get the error category.

```
const error_category & category() const;
```

5.117.2 error_code::error_code

Default constructor.

```
error_code();
```

Construct with specific error code and category.

```
error_code(
    int v,
    const error_category & c);
```

Construct from an error code enum.

```
template<
    typename ErrorEnum>
error_code(
    ErrorEnum e);
```

5.117.2.1 error_code::error_code (1 of 3 overloads)

Default constructor.

```
error_code();
```

5.117.2.2 error_code::error_code (2 of 3 overloads)

Construct with specific error code and category.

```
error_code(
    int v,
    const error_category & c);
```

5.117.2.3 error_code::error_code (3 of 3 overloads)

Construct from an error code enum.

```
template<
    typename ErrorEnum>
error_code(
    ErrorEnum e);
```

5.117.3 error_code::message

Get the message associated with the error.

```
std::string message() const;
```

5.117.4 error_code::operator unspecified_bool_type

Operator returns non-null if there is a non-success error code.

```
operator unspecified_bool_type() const;
```

5.117.5 error_code::operator!

Operator to test if the error represents success.

```
bool operator!() const;
```

5.117.6 error_code::operator!=

Inequality operator to compare two error objects.

```
friend bool operator!=(  
    const error_code & e1,  
    const error_code & e2);
```

Requirements

Header: asio/error_code.hpp

Convenience header: asio.hpp

5.117.7 error_code::operator==

Equality operator to compare two error objects.

```
friend bool operator==(  
    const error_code & e1,  
    const error_code & e2);
```

Requirements

Header: asio/error_code.hpp

Convenience header: asio.hpp

5.117.8 error_code::unspecified_bool_true

```
static void unspecified_bool_true(  
    unspecified_bool_type_t );
```

5.117.9 error_code::unspecified_bool_type

```
typedef void(*) unspecified_bool_type;
```

Requirements

Header: asio/error_code.hpp

Convenience header: asio.hpp

5.117.10 `error_code::value`

Get the error value.

```
int value() const;
```

5.118 `error_code::unspecified_bool_type_t`

```
struct unspecified_bool_type_t
```

Requirements

Header: asio/error_code.hpp

Convenience header: asio.hpp

5.119 `generic::basic_endpoint`

Describes an endpoint for any socket type.

```
template<
    typename Protocol>
class basic_endpoint
```

Types

Name	Description
<code>data_type</code>	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
<code>protocol_type</code>	The protocol type associated with the endpoint.

Member Functions

Name	Description
<code>basic_endpoint</code>	Default constructor. Construct an endpoint from the specified socket address. Construct an endpoint from the specific endpoint type. Copy constructor.
<code>capacity</code>	Get the capacity of the endpoint in the native type.
<code>data</code>	Get the underlying endpoint in the native type.
<code>operator=</code>	Assign from another endpoint.
<code>protocol</code>	The protocol associated with the endpoint.

Name	Description
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

The `generic::basic_endpoint` class template describes an endpoint that may be associated with any socket type.

Remarks

The socket types `sockaddr` type must be able to fit into a `sockaddr_storage` structure.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/generic/basic_endpoint.hpp`

Convenience header: `asio.hpp`

5.119.1 generic::basic_endpoint::basic_endpoint

Default constructor.

```
basic_endpoint();
```

Construct an endpoint from the specified socket address.

```
basic_endpoint(
    const void * socket_address,
    std::size_t socket_address_size,
    int socket_protocol = 0);
```

Construct an endpoint from the specific endpoint type.

```
template<
    typename Endpoint>
basic_endpoint(
    const Endpoint & endpoint);
```

Copy constructor.

```
basic_endpoint(
    const basic_endpoint & other);
```

5.119.1.1 generic::basic_endpoint::basic_endpoint (1 of 4 overloads)

Default constructor.

```
basic_endpoint();
```

5.119.1.2 generic::basic_endpoint::basic_endpoint (2 of 4 overloads)

Construct an endpoint from the specified socket address.

```
basic_endpoint(
    const void * socket_address,
    std::size_t socket_address_size,
    int socket_protocol = 0);
```

5.119.1.3 generic::basic_endpoint::basic_endpoint (3 of 4 overloads)

Construct an endpoint from the specific endpoint type.

```
template<
    typename Endpoint>
basic_endpoint(
    const Endpoint & endpoint);
```

5.119.1.4 generic::basic_endpoint::basic_endpoint (4 of 4 overloads)

Copy constructor.

```
basic_endpoint(
    const basic_endpoint & other);
```

5.119.2 generic::basic_endpoint::capacity

Get the capacity of the endpoint in the native type.

```
std::size_t capacity() const;
```

5.119.3 generic::basic_endpoint::data

Get the underlying endpoint in the native type.

```
data_type * data();
```

```
const data_type * data() const;
```

5.119.3.1 generic::basic_endpoint::data (1 of 2 overloads)

Get the underlying endpoint in the native type.

```
data_type * data();
```

5.119.3.2 generic::basic_endpoint::data (2 of 2 overloads)

Get the underlying endpoint in the native type.

```
const data_type * data() const;
```

5.119.4 generic::basic_endpoint::data_type

The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.

```
typedef implementation_defined data_type;
```

Requirements

Header: asio/generic/basic_endpoint.hpp

Convenience header: asio.hpp

5.119.5 generic::basic_endpoint::operator!=

Compare two endpoints for inequality.

```
friend bool operator!=(
    const basic_endpoint< Protocol > & e1,
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/generic/basic_endpoint.hpp

Convenience header: asio.hpp

5.119.6 generic::basic_endpoint::operator<

Compare endpoints for ordering.

```
friend bool operator<(
    const basic_endpoint< Protocol > & e1,
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/generic/basic_endpoint.hpp

Convenience header: asio.hpp

5.119.7 generic::basic_endpoint::operator<=

Compare endpoints for ordering.

```
friend bool operator<=
    const basic_endpoint< Protocol > & e1,
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/generic/basic_endpoint.hpp

Convenience header: asio.hpp

5.119.8 generic::basic_endpoint::operator=

Assign from another endpoint.

```
basic_endpoint & operator=(
    const basic_endpoint & other);
```

5.119.9 generic::basic_endpoint::operator==

Compare two endpoints for equality.

```
friend bool operator==(

    const basic_endpoint< Protocol > & e1,
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/generic/basic_endpoint.hpp

Convenience header: asio.hpp

5.119.10 generic::basic_endpoint::operator>

Compare endpoints for ordering.

```
friend bool operator>(
    const basic_endpoint< Protocol > & e1,
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/generic/basic_endpoint.hpp

Convenience header: asio.hpp

5.119.11 generic::basic_endpoint::operator>=

Compare endpoints for ordering.

```
friend bool operator>=
  const basic_endpoint< Protocol > & e1,
  const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/generic/basic_endpoint.hpp

Convenience header: asio.hpp

5.119.12 generic::basic_endpoint::protocol

The protocol associated with the endpoint.

```
protocol_type protocol() const;
```

5.119.13 generic::basic_endpoint::protocol_type

The protocol type associated with the endpoint.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/generic/basic_endpoint.hpp

Convenience header: asio.hpp

5.119.14 generic::basic_endpoint::resize

Set the underlying size of the endpoint in the native type.

```
void resize(
  std::size_t new_size);
```

5.119.15 generic::basic_endpoint::size

Get the underlying size of the endpoint in the native type.

```
std::size_t size() const;
```

5.120 generic::datagram_protocol

Encapsulates the flags needed for a generic datagram-oriented socket.

```
class datagram_protocol
```

Types

Name	Description
endpoint	The type of an endpoint.
socket	The generic socket type.

Member Functions

Name	Description
datagram_protocol	Construct a protocol object for a specific address family and protocol. Construct a generic protocol object from a specific protocol.
family	Obtain an identifier for the protocol family.
protocol	Obtain an identifier for the protocol.
type	Obtain an identifier for the type of the protocol.

Friends

Name	Description
operator!=	Compare two protocols for inequality.
operator==	Compare two protocols for equality.

The `generic::datagram_protocol` class contains flags necessary for datagram-oriented sockets of any address family and protocol.

Examples

Constructing using a native address family and socket protocol:

```
datagram_protocol p(AF_INET, IPPROTO_UDP);
```

Constructing from a specific protocol type:

```
datagram_protocol p(asio::ip::udp::v4());
```

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: `asio/generic/datagram_protocol.hpp`

Convenience header: `asio.hpp`

5.120.1 generic::datagram_protocol::datagram_protocol

Construct a protocol object for a specific address family and protocol.

```
datagram_protocol(
    int address_family,
    int socket_protocol);
```

Construct a generic protocol object from a specific protocol.

```
template<
    typename Protocol>
datagram_protocol(
    const Protocol & source_protocol);
```

5.120.1.1 generic::datagram_protocol::datagram_protocol (1 of 2 overloads)

Construct a protocol object for a specific address family and protocol.

```
datagram_protocol(
    int address_family,
    int socket_protocol);
```

5.120.1.2 generic::datagram_protocol::datagram_protocol (2 of 2 overloads)

Construct a generic protocol object from a specific protocol.

```
template<
    typename Protocol>
datagram_protocol(
    const Protocol & source_protocol);
```

Exceptions

@c bad_cast Thrown if the source protocol is not datagram-oriented.

5.120.2 generic::datagram_protocol::endpoint

The type of an endpoint.

```
typedef basic_endpoint< datagram_protocol > endpoint;
```

Types

Name	Description
data_type	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
protocol_type	The protocol type associated with the endpoint.

Member Functions

Name	Description
basic_endpoint	Default constructor. Construct an endpoint from the specified socket address. Construct an endpoint from the specific endpoint type. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

The `generic::basic_endpoint` class template describes an endpoint that may be associated with any socket type.

Remarks

The socket types `sockaddr` type must be able to fit into a `sockaddr_storage` structure.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/generic/datagram_protocol.hpp`

Convenience header: `asio.hpp`

5.120.3 generic::datagram_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.120.4 generic::datagram_protocol::operator!=

Compare two protocols for inequality.

```
friend bool operator!=(
    const datagram_protocol & p1,
    const datagram_protocol & p2);
```

Requirements

Header: asio/generic/datagram_protocol.hpp

Convenience header: asio.hpp

5.120.5 generic::datagram_protocol::operator==

Compare two protocols for equality.

```
friend bool operator==((
    const datagram_protocol & p1,
    const datagram_protocol & p2);
```

Requirements

Header: asio/generic/datagram_protocol.hpp

Convenience header: asio.hpp

5.120.6 generic::datagram_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.120.7 generic::datagram_protocol::socket

The generic socket type.

```
typedef basic_datagram_socket< datagram_protocol > socket;
```

Name	Description
------	-------------

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.

Name	Description
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive on a connected socket.
async_receive_from	Start an asynchronous receive.
async_send	Start an asynchronous send on a connected socket.
async_send_to	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_datagram_socket	Construct a basic_datagram_socket without opening it. Construct and open a basic_datagram_socket. Construct a basic_datagram_socket, opening it and binding it to the given local endpoint. Construct a basic_datagram_socket on an existing native socket. Move-construct a basic_datagram_socket from another. Move-construct a basic_datagram_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.

Name	Description
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.
operator=	Move-assign a basic_datagram_socket from another. Move-assign a basic_datagram_socket from a socket of another protocol type.
receive	Receive some data on a connected socket.
receive_from	Receive a datagram with the endpoint of the sender.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on a connected socket.
send_to	Send a datagram to the specified endpoint.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.

Name	Description
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_datagram_socket` class template provides asynchronous and blocking datagram-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/generic/datagram_protocol.hpp`

Convenience header: `asio.hpp`

5.120.8 generic::datagram_protocol::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.121 generic::raw_protocol

Encapsulates the flags needed for a generic raw socket.

```
class raw_protocol
```

Types

Name	Description
endpoint	The type of an endpoint.
socket	The generic socket type.

Member Functions

Name	Description
family	Obtain an identifier for the protocol family.
protocol	Obtain an identifier for the protocol.
raw_protocol	Construct a protocol object for a specific address family and protocol. Construct a generic protocol object from a specific protocol.
type	Obtain an identifier for the type of the protocol.

Friends

Name	Description
operator!=	Compare two protocols for inequality.
operator==	Compare two protocols for equality.

The `generic::raw_protocol` class contains flags necessary for raw sockets of any address family and protocol.

Examples

Constructing using a native address family and socket protocol:

```
raw_protocol p(AF_INET, IPPROTO_ICMP);
```

Constructing from a specific protocol type:

```
raw_protocol p(asio::ip::icmp::v4());
```

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: `asio/generic/raw_protocol.hpp`

Convenience header: `asio.hpp`

5.121.1 generic::raw_protocol::endpoint

The type of an endpoint.

```
typedef basic_endpoint< raw_protocol > endpoint;
```

Types

Name	Description
data_type	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
protocol_type	The protocol type associated with the endpoint.

Member Functions

Name	Description
basic_endpoint	Default constructor. Construct an endpoint from the specified socket address. Construct an endpoint from the specific endpoint type. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.

Name	Description
operator>=	Compare endpoints for ordering.

The `generic::basic_endpoint` class template describes an endpoint that may be associated with any socket type.

Remarks

The socket types `sockaddr` type must be able to fit into a `sockaddr_storage` structure.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/generic/raw_protocol.hpp`

Convenience header: `asio.hpp`

5.121.2 generic::raw_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.121.3 generic::raw_protocol::operator!=

Compare two protocols for inequality.

```
friend bool operator!=(
    const raw_protocol & p1,
    const raw_protocol & p2);
```

Requirements

Header: `asio/generic/raw_protocol.hpp`

Convenience header: `asio.hpp`

5.121.4 generic::raw_protocol::operator==

Compare two protocols for equality.

```
friend bool operator==((
    const raw_protocol & p1,
    const raw_protocol & p2);
```

Requirements

Header: asio/generic/raw_protocol.hpp

Convenience header: asio.hpp

5.121.5 generic::raw_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.121.6 generic::raw_protocol::raw_protocol

Construct a protocol object for a specific address family and protocol.

```
raw_protocol(
    int address_family,
    int socket_protocol);
```

Construct a generic protocol object from a specific protocol.

```
template<
    typename Protocol>
raw_protocol(
    const Protocol & source_protocol);
```

5.121.6.1 generic::raw_protocol::raw_protocol (1 of 2 overloads)

Construct a protocol object for a specific address family and protocol.

```
raw_protocol(
    int address_family,
    int socket_protocol);
```

5.121.6.2 generic::raw_protocol::raw_protocol (2 of 2 overloads)

Construct a generic protocol object from a specific protocol.

```
template<
    typename Protocol>
raw_protocol(
    const Protocol & source_protocol);
```

Exceptions

@c bad_cast Thrown if the source protocol is not raw-oriented.

5.121.7 generic::raw_protocol::socket

The generic socket type.

```
typedef basic_raw_socket< raw_protocol > socket;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive on a connected socket.
async_receive_from	Start an asynchronous receive.
async_send	Start an asynchronous send on a connected socket.
async_send_to	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_raw_socket	Construct a basic_raw_socket without opening it. Construct and open a basic_raw_socket. Construct a basic_raw_socket, opening it and binding it to the given local endpoint. Construct a basic_raw_socket on an existing native socket. Move-construct a basic_raw_socket from another. Move-construct a basic_raw_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.

Name	Description
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.
operator=	Move-assign a basic_raw_socket from another. Move-assign a basic_raw_socket from a socket of another protocol type.
receive	Receive some data on a connected socket.
receive_from	Receive raw data with the endpoint of the sender.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on a connected socket.
send_to	Send raw data to the specified endpoint.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.

Name	Description
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_raw_socket` class template provides asynchronous and blocking raw-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/generic/raw_protocol.hpp`

Convenience header: `asio.hpp`

5.121.8 generic::raw_protocol::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.122 generic::seq_packet_protocol

Encapsulates the flags needed for a generic sequenced packet socket.

```
class seq_packet_protocol
```

Types

Name	Description
endpoint	The type of an endpoint.
socket	The generic socket type.

Member Functions

Name	Description
family	Obtain an identifier for the protocol family.
protocol	Obtain an identifier for the protocol.
seq_packet_protocol	Construct a protocol object for a specific address family and protocol. Construct a generic protocol object from a specific protocol.
type	Obtain an identifier for the type of the protocol.

Friends

Name	Description
operator!=	Compare two protocols for inequality.
operator==	Compare two protocols for equality.

The `generic::seq_packet_protocol` class contains flags necessary for seq_packet-oriented sockets of any address family and protocol.

Examples

Constructing using a native address family and socket protocol:

```
seq_packet_protocol p(AF_INET, IPPROTO_SCTP);
```

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: `asio/generic/seq_packet_protocol.hpp`

Convenience header: `asio.hpp`

5.122.1 `generic::seq_packet_protocol::endpoint`

The type of an endpoint.

```
typedef basic_endpoint< seq_packet_protocol > endpoint;
```

Types

Name	Description
data_type	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
protocol_type	The protocol type associated with the endpoint.

Member Functions

Name	Description
basic_endpoint	Default constructor. Construct an endpoint from the specified socket address. Construct an endpoint from the specific endpoint type. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

The `generic::basic_endpoint` class template describes an endpoint that may be associated with any socket type.

Remarks

The socket types `sockaddr` type must be able to fit into a `sockaddr_storage` structure.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/generic/seq_packet_protocol.hpp

Convenience header: asio.hpp

5.122.2 generic::seq_packet_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.122.3 generic::seq_packet_protocol::operator!=

Compare two protocols for inequality.

```
friend bool operator!=(
    const seq_packet_protocol & p1,
    const seq_packet_protocol & p2);
```

Requirements

Header: asio/generic/seq_packet_protocol.hpp

Convenience header: asio.hpp

5.122.4 generic::seq_packet_protocol::operator==

Compare two protocols for equality.

```
friend bool operator==((
    const seq_packet_protocol & p1,
    const seq_packet_protocol & p2);
```

Requirements

Header: asio/generic/seq_packet_protocol.hpp

Convenience header: asio.hpp

5.122.5 generic::seq_packet_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.122.6 generic::seq_packet_protocol::seq_packet_protocol

Construct a protocol object for a specific address family and protocol.

```
seq_packet_protocol(
    int address_family,
    int socket_protocol);
```

Construct a generic protocol object from a specific protocol.

```
template<
    typename Protocol>
seq_packet_protocol(
    const Protocol & source_protocol);
```

5.122.6.1 generic::seq_packet_protocol::seq_packet_protocol (1 of 2 overloads)

Construct a protocol object for a specific address family and protocol.

```
seq_packet_protocol(
    int address_family,
    int socket_protocol);
```

5.122.6.2 generic::seq_packet_protocol::seq_packet_protocol (2 of 2 overloads)

Construct a generic protocol object from a specific protocol.

```
template<
    typename Protocol>
seq_packet_protocol(
    const Protocol & source_protocol);
```

Exceptions

@c bad_cast Thrown if the source protocol is not based around sequenced packets.

5.122.7 generic::seq_packet_protocol::socket

The generic socket type.

```
typedef basic_seq_packet_socket< seq_packet_protocol > socket;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.

Name	Description
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.

Name	Description
async_receive	Start an asynchronous receive.
async_send	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_seq_packet_socket	Construct a basic_seq_packet_socket without opening it. Construct and open a basic_seq_packet_socket. Construct a basic_seq_packet_socket, opening it and binding it to the given local endpoint. Construct a basic_seq_packet_socket on an existing native socket. Move-construct a basic_seq_packet_socket from another. Move-construct a basic_seq_packet_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.

Name	Description
operator=	Move-assign a basic_seq_packet_socket from another. Move-assign a basic_seq_packet_socket from a socket of another protocol type.
receive	Receive some data on the socket. Receive some data on a connected socket.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on the socket.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.

Name	Description
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_seq_packet_socket` class template provides asynchronous and blocking sequenced packet socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/generic/seq_packet_protocol.hpp`

Convenience header: `asio.hpp`

5.122.8 `generic::seq_packet_protocol::type`

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.123 `generic::stream_protocol`

Encapsulates the flags needed for a generic stream-oriented socket.

```
class stream_protocol
```

Types

Name	Description
endpoint	The type of an endpoint.
iostream	The generic socket iostream type.
socket	The generic socket type.

Member Functions

Name	Description
family	Obtain an identifier for the protocol family.
protocol	Obtain an identifier for the protocol.

Name	Description
stream_protocol	Construct a protocol object for a specific address family and protocol. Construct a generic protocol object from a specific protocol.
type	Obtain an identifier for the type of the protocol.

Friends

Name	Description
operator!=	Compare two protocols for inequality.
operator==	Compare two protocols for equality.

The `generic::stream_protocol` class contains flags necessary for stream-oriented sockets of any address family and protocol.

Examples

Constructing using a native address family and socket protocol:

```
stream_protocol p(AF_INET, IPPROTO_TCP);
```

Constructing from a specific protocol type:

```
stream_protocol p(asio::ip::tcp::v4());
```

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: asio/generic/stream_protocol.hpp

Convenience header: asio.hpp

5.123.1 generic::stream_protocol::endpoint

The type of an endpoint.

```
typedef basic_endpoint< stream_protocol > endpoint;
```

Types

Name	Description
data_type	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
protocol_type	The protocol type associated with the endpoint.

Member Functions

Name	Description
basic_endpoint	Default constructor. Construct an endpoint from the specified socket address. Construct an endpoint from the specific endpoint type. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

The `generic::basic_endpoint` class template describes an endpoint that may be associated with any socket type.

Remarks

The socket types `sockaddr` type must be able to fit into a `sockaddr_storage` structure.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/generic/stream_protocol.hpp

Convenience header: asio.hpp

5.123.2 generic::stream_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.123.3 generic::stream_protocol::iostream

The generic socket iostream type.

```
typedef basic_socket_iostream< stream_protocol > iostream;
```

Types

Name	Description
duration_type	The duration type.
endpoint_type	The endpoint type.
time_type	The time type.

Member Functions

Name	Description
basic_socket_iostream	Construct a basic_socket_iostream without establishing a connection. Establish a connection to an endpoint corresponding to a resolver query.
close	Close the connection.
connect	Establish a connection to an endpoint corresponding to a resolver query.
error	Get the last error associated with the stream.
expires_at	Get the stream's expiry time as an absolute time. Set the stream's expiry time as an absolute time.

Name	Description
<code>expires_from_now</code>	Get the timer's expiry time relative to now. Set the stream's expiry time relative to now.
<code>rdbuf</code>	Return a pointer to the underlying <code>streambuf</code> .

Requirements

Header: `asio/generic/stream_protocol.hpp`

Convenience header: `asio.hpp`

5.123.4 `generic::stream_protocol::operator!=`

Compare two protocols for inequality.

```
friend bool operator!=(
    const stream_protocol & p1,
    const stream_protocol & p2);
```

Requirements

Header: `asio/generic/stream_protocol.hpp`

Convenience header: `asio.hpp`

5.123.5 `generic::stream_protocol::operator==`

Compare two protocols for equality.

```
friend bool operator==(

    const stream_protocol & p1,
    const stream_protocol & p2);
```

Requirements

Header: `asio/generic/stream_protocol.hpp`

Convenience header: `asio.hpp`

5.123.6 `generic::stream_protocol::protocol`

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.123.7 `generic::stream_protocol::socket`

The generic socket type.

```
typedef basic_stream_socket< stream_protocol > socket;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_read_some	Start an asynchronous read.
async_receive	Start an asynchronous receive.
async_send	Start an asynchronous send.
async_write_some	Start an asynchronous write.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_stream_socket	Construct a basic_stream_socket without opening it. Construct and open a basic_stream_socket. Construct a basic_stream_socket, opening it and binding it to the given local endpoint. Construct a basic_stream_socket on an existing native socket. Move-construct a basic_stream_socket from another. Move-construct a basic_stream_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.

Name	Description
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.
operator=	Move-assign a basic_stream_socket from another. Move-assign a basic_stream_socket from a socket of another protocol type.
read_some	Read some data from the socket.
receive	Receive some data on the socket. Receive some data on a connected socket.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on the socket.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.
write_some	Write some data to the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.

Name	Description
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_stream_socket` class template provides asynchronous and blocking stream-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/generic/stream_protocol.hpp`

Convenience header: `asio.hpp`

5.123.8 `generic::stream_protocol::stream_protocol`

Construct a protocol object for a specific address family and protocol.

```
stream_protocol(
    int address_family,
    int socket_protocol);
```

Construct a generic protocol object from a specific protocol.

```
template<
    typename Protocol>
stream_protocol(
    const Protocol & source_protocol);
```

5.123.8.1 `generic::stream_protocol::stream_protocol (1 of 2 overloads)`

Construct a protocol object for a specific address family and protocol.

```
stream_protocol(
    int address_family,
    int socket_protocol);
```

5.123.8.2 generic::stream_protocol::stream_protocol (2 of 2 overloads)

Construct a generic protocol object from a specific protocol.

```
template<
    typename Protocol>
stream_protocol(
    const Protocol & source_protocol);
```

Exceptions

@c bad_cast Thrown if the source protocol is not stream-oriented.

5.123.9 generic::stream_protocol::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.124 handler_type

Default handler type traits provided for all handlers.

```
template<
    typename Handler,
    typename Signature>
struct handler_type
```

Types

Name	Description
type	The handler type for the specific signature.

The `handler_type` traits class is used for determining the concrete handler type to be used for an asynchronous operation. It allows the handler type to be determined at the point where the specific completion handler signature is known.

This template may be specialised for user-defined handler types.

Requirements

Header: asio/handler_type.hpp

Convenience header: asio.hpp

5.124.1 handler_type::type

The handler type for the specific signature.

```
typedef Handler type;
```

Requirements

Header: asio/handler_type.hpp

Convenience header: asio.hpp

5.125 has_service

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

This function is used to determine whether the `io_service` contains a service object corresponding to the given service type.

Parameters

`ios` The `io_service` object that owns the service.

Return Value

A boolean indicating whether the `io_service` contains the service.

Requirements

Header: asio/io_service.hpp

Convenience header: asio.hpp

5.126 high_resolution_timer

Typedef for a timer based on the high resolution clock.

```
typedef basic_waitable_timer< chrono::high_resolution_clock > high_resolution_timer;
```

Types

Name	Description
<code>clock_type</code>	The clock type.
<code>duration</code>	The duration type of the clock.
<code>implementation_type</code>	The underlying implementation type of I/O object.
<code>service_type</code>	The type of the service that will be used to provide I/O operations.
<code>time_point</code>	The time point type of the clock.
<code>traits_type</code>	The wait traits type.

Member Functions

Name	Description
<code>async_wait</code>	Start an asynchronous wait on the timer.
<code>basic_writable_timer</code>	Constructor. Constructor to set a particular expiry time as an absolute time. Constructor to set a particular expiry time relative to now.
<code>cancel</code>	Cancel any asynchronous operations that are waiting on the timer.
<code>cancel_one</code>	Cancels one asynchronous operation that is waiting on the timer.
<code>expires_at</code>	Get the timer's expiry time as an absolute time. Set the timer's expiry time as an absolute time.
<code>expires_from_now</code>	Get the timer's expiry time relative to now. Set the timer's expiry time relative to now.
<code>get_io_service</code>	Get the io_service associated with the object.
<code>wait</code>	Perform a blocking wait on the timer.

Protected Member Functions

Name	Description
<code>get_implementation</code>	Get the underlying implementation of the I/O object.
<code>get_service</code>	Get the service associated with the I/O object.

Protected Data Members

Name	Description
<code>implementation</code>	(Deprecated: Use <code>get_implementation()</code>) The underlying implementation of the I/O object.
<code>service</code>	(Deprecated: Use <code>get_service()</code>) The service associated with the I/O object.

The `basic_writable_timer` class template provides the ability to perform a blocking or asynchronous wait for a timer to expire.

A writable timer is always in one of two states: "expired" or "not expired". If the `wait()` or `async_wait()` function is called on an expired timer, the wait operation will complete immediately.

Most applications will use one of the `steady_timer`, `system_timer` or `high_resolution_timer` typedefs.

Remarks

This waitable timer functionality is for use with the C++11 standard library's `<chrono>` facility, or with the Boost.Chrono library.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Examples

Performing a blocking wait (C++11):

```
// Construct a timer without setting an expiry time.  
asio::steady_timer timer(io_service);  
  
// Set an expiry time relative to now.  
timer.expires_from_now(std::chrono::seconds(5));  
  
// Wait for the timer to expire.  
timer.wait();
```

Performing an asynchronous wait (C++11):

```
void handler(const asio::error_code& error)  
{  
    if (!error)  
    {  
        // Timer expired.  
    }  
}  
  
...  
  
// Construct a timer with an absolute expiry time.  
asio::steady_timer timer(io_service,  
    std::chrono::steady_clock::now() + std::chrono::seconds(60));  
  
// Start an asynchronous wait.  
timer.async_wait(handler);
```

Changing an active waitable timer's expiry time

Changing the expiry time of a timer while there are pending asynchronous waits causes those wait operations to be cancelled. To ensure that the action associated with the timer is performed only once, use something like this: used:

```
void on_some_event()  
{  
    if (my_timer.expires_from_now(seconds(5)) > 0)  
    {  
        // We managed to cancel the timer. Start new asynchronous wait.  
        my_timer.async_wait(on_timeout);  
    }  
    else  
    {  
        // Too late, timer has already expired!  
    }  
}
```

```

void on_timeout(const asio::error_code& e)
{
    if (e != asio::error::operation_aborted)
    {
        // Timer was not cancelled, take necessary action.
    }
}

```

- The `asio::basic_waitable_timer::expires_from_now()` function cancels any pending asynchronous waits, and returns the number of asynchronous waits that were cancelled. If it returns 0 then you were too late and the wait handler has already been executed, or will soon be executed. If it returns 1 then the wait handler was successfully cancelled.
- If a wait handler is cancelled, the `error_code` passed to it contains the value `asio::error::operation_aborted`.

This typedef uses the C++11 `<chrono>` standard library facility, if available. Otherwise, it may use the Boost.Chrono library. To explicitly utilise Boost.Chrono, use the `basic_waitable_timer` template directly:

```
typedef basic_waitable_timer<boost::chrono::high_resolution_clock> timer;
```

Requirements

Header: `asio/high_resolution_timer.hpp`

Convenience header: None

5.127 invalid_service_owner

Exception thrown when trying to add a service object to an `io_service` where the service has a different owner.

```
class invalid_service_owner
```

Member Functions

Name	Description
<code>invalid_service_owner</code>	

Requirements

Header: `asio/io_service.hpp`

Convenience header: `asio.hpp`

5.127.1 invalid_service_owner::invalid_service_owner

```
invalid_service_owner();
```

5.128 io_service

Provides core I/O functionality.

```
class io_service :  
    noncopyable
```

Types

Name	Description
id	Class used to uniquely identify a service.
service	Base class for all io_service services.
strand	Provides serialised handler execution.
work	Class to inform the io_service when it has work to do.
fork_event	Fork-related event notifications.

Member Functions

Name	Description
dispatch	Request the io_service to invoke the given handler.
io_service	Constructor.
notify_fork	Notify the io_service of a fork-related event.
poll	Run the io_service object's event processing loop to execute ready handlers.
poll_one	Run the io_service object's event processing loop to execute one ready handler.
post	Request the io_service to invoke the given handler and return immediately.
reset	Reset the io_service in preparation for a subsequent run() invocation.
run	Run the io_service object's event processing loop.
run_one	Run the io_service object's event processing loop to execute at most one handler.
stop	Stop the io_service object's event processing loop.
stopped	Determine whether the io_service object has been stopped.
wrap	Create a new handler that automatically dispatches the wrapped handler on the io_service.
~io_service	Destructor.

Friends

Name	Description
add_service	Add a service object to the io_service.
has_service	Determine if an io_service contains a specified service type.
use_service	Obtain the service object corresponding to the given type.

The `io_service` class provides the core I/O functionality for users of the asynchronous I/O objects, including:

- `asio::ip::tcp::socket`
- `asio::ip::tcp::acceptor`
- `asio::ip::udp::socket`
- `deadline_timer`.

The `io_service` class also includes facilities intended for developers of custom asynchronous services.

Thread Safety

Distinct objects: Safe.

Shared objects: Safe, with the specific exceptions of the `reset()` and `notify_fork()` functions. Calling `reset()` while there are unfinished `run()`, `run_one()`, `poll()` or `poll_one()` calls results in undefined behaviour. The `notify_fork()` function should not be called while any `io_service` function, or any function on an I/O object that is associated with the `io_service`, is being called in another thread.

Synchronous and asynchronous operations

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`.

Effect of exceptions thrown from handlers

If an exception is thrown from a handler, the exception is allowed to propagate through the throwing thread's invocation of `run()`, `run_one()`, `poll()` or `poll_one()`. No other threads that are calling any of these functions are affected. It is then the responsibility of the application to catch the exception.

After the exception has been caught, the `run()`, `run_one()`, `poll()` or `poll_one()` call may be restarted *without* the need for an intervening call to `reset()`. This allows the thread to rejoin the `io_service` object's thread pool without impacting any other threads in the pool.

For example:

```
asio::io_service io_service;
...
for (;;)
{
    try
```

```

    {
        io_service.run();
        break; // run() exited normally
    }
    catch (my_exception& e)
    {
        // Deal with exception as appropriate.
    }
}

```

Stopping the `io_service` from running out of work

Some applications may need to prevent an `io_service` object's `run()` call from returning when there is no more work to do. For example, the `io_service` may be being run in a background thread that is launched prior to the application's asynchronous operations. The `run()` call may be kept running by creating an object of type `io_service::work`:

```

asio::io_service io_service;
asio::io_service::work work(io_service);
...

```

To effect a shutdown, the application will then need to call the `io_service` object's `stop()` member function. This will cause the `io_service` `run()` call to return as soon as possible, abandoning unfinished operations and without permitting ready handlers to be dispatched.

Alternatively, if the application requires that all operations and handlers be allowed to finish normally, the `work` object may be explicitly destroyed.

```

asio::io_service io_service;
auto_ptr<asio::io_service::work> work(
    new asio::io_service::work(io_service));
...
work.reset(); // Allow run() to exit.

```

The `io_service` class and I/O services

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.

I/O services exist to manage the logical interface to the operating system on behalf of the I/O objects. In particular, there are resources that are shared across a class of I/O objects. For example, timers may be implemented in terms of a single timer queue. The I/O services manage these shared resources.

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.

All I/O service implementations have `io_service::service` as a public base class. Custom I/O services may be implemented by deriving from this class and then added to an `io_service` using the facilities described above.

Requirements

Header: asio/io_service.hpp

Convenience header: asio.hpp

5.128.1 io_service::add_service

Add a service object to the **io_service**.

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

This function is used to add a service to the **io_service**.

Parameters

ios The **io_service** object that owns the service.

svc The service object. On success, ownership of the service object is transferred to the **io_service**. When the **io_service** object is destroyed, it will destroy the service object by performing:

```
delete static_cast<io_service::service*>(svc)
```

Exceptions

asio::service_already_exists Thrown if a service of the given type is already present in the **io_service**.

asio::invalid_service_owner Thrown if the service's owning **io_service** is not the **io_service** object specified by the **ios** parameter.

Requirements

Header: asio/io_service.hpp

Convenience header: asio.hpp

5.128.2 io_service::dispatch

Request the **io_service** to invoke the given handler.

```
template<
    typename CompletionHandler>
void-or-deduced dispatch(
    CompletionHandler handler);
```

This function is used to ask the **io_service** to execute the given handler.

The **io_service** guarantees that the handler will only be called in a thread in which the `run()`, `run_one()`, `poll()` or `poll_one()` member functions are currently being invoked. The handler may be executed inside this function if the guarantee can be met.

Parameters

handler The handler to be called. The `io_service` will make a copy of the handler object as required. The function signature of the handler must be:

```
void handler();
```

Remarks

This function throws an exception only if:

- the handler's `asio_handler_allocate` function; or
- the handler's copy constructor

throws an exception.

5.128.3 `io_service::fork_event`

Fork-related event notifications.

```
enum fork_event
```

Values

fork_prepare Notify the `io_service` that the process is about to fork.

fork_parent Notify the `io_service` that the process has forked and is the parent.

fork_child Notify the `io_service` that the process has forked and is the child.

5.128.4 `io_service::has_service`

Determine if an `io_service` contains a specified service type.

```
template<
    typename Service>
friend bool has_service(
    io_service & ios);
```

This function is used to determine whether the `io_service` contains a service object corresponding to the given service type.

Parameters

ios The `io_service` object that owns the service.

Return Value

A boolean indicating whether the `io_service` contains the service.

Requirements

Header: asio/io_service.hpp

Convenience header: asio.hpp

5.128.5 io_service::io_service

Constructor.

```
io_service();  
  
explicit io_service(  
    std::size_t concurrency_hint);
```

5.128.5.1 io_service::io_service (1 of 2 overloads)

Constructor.

```
io_service();
```

5.128.5.2 io_service::io_service (2 of 2 overloads)

Constructor.

```
io_service(  
    std::size_t concurrency_hint);
```

Construct with a hint about the required level of concurrency.

Parameters

concurrency_hint A suggestion to the implementation on how many threads it should allow to run simultaneously.

5.128.6 io_service::notify_fork

Notify the **io_service** of a fork-related event.

```
void notify_fork(  
    asio::io_service::fork_event event);
```

This function is used to inform the **io_service** that the process is about to fork, or has just forked. This allows the **io_service**, and the services it contains, to perform any necessary housekeeping to ensure correct operation following a fork.

This function must not be called while any other **io_service** function, or any function on an I/O object associated with the **io_service**, is being called in another thread. It is, however, safe to call this function from within a completion handler, provided no other thread is accessing the **io_service**.

Parameters

event A fork-related event.

Exceptions

asio::system_error Thrown on failure. If the notification fails the **io_service** object should no longer be used and should be destroyed.

Example

The following code illustrates how to incorporate the `notify_fork()` function:

```
my_io_service.notify_fork(asio::io_service::fork_prepare);
if (fork() == 0)
{
    // This is the child process.
    my_io_service.notify_fork(asio::io_service::fork_child);
}
else
{
    // This is the parent process.
    my_io_service.notify_fork(asio::io_service::fork_parent);
}
```

Remarks

For each service object `svc` in the **io_service** set, performs `svc->fork_service()`. When processing the `fork_prepare` event, services are visited in reverse order of the beginning of service object lifetime. Otherwise, services are visited in order of the beginning of service object lifetime.

5.128.7 **io_service::poll**

Run the **io_service** object's event processing loop to execute ready handlers.

```
std::size_t poll();

std::size_t poll(
    asio::error_code & ec);
```

5.128.7.1 **io_service::poll (1 of 2 overloads)**

Run the **io_service** object's event processing loop to execute ready handlers.

```
std::size_t poll();
```

The `poll()` function runs handlers that are ready to run, without blocking, until the **io_service** has been stopped or there are no more ready handlers.

Return Value

The number of handlers that were executed.

Exceptions

asio::system_error Thrown on failure.

5.128.7.2 `io_service::poll` (2 of 2 overloads)

Run the `io_service` object's event processing loop to execute ready handlers.

```
std::size_t poll(
    asio::error_code & ec);
```

The `poll()` function runs handlers that are ready to run, without blocking, until the `io_service` has been stopped or there are no more ready handlers.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of handlers that were executed.

5.128.8 `io_service::poll_one`

Run the `io_service` object's event processing loop to execute one ready handler.

```
std::size_t poll_one();
std::size_t poll_one(
    asio::error_code & ec);
```

5.128.8.1 `io_service::poll_one` (1 of 2 overloads)

Run the `io_service` object's event processing loop to execute one ready handler.

```
std::size_t poll_one();
```

The `poll_one()` function runs at most one handler that is ready to run, without blocking.

Return Value

The number of handlers that were executed.

Exceptions

`asio::system_error` Thrown on failure.

5.128.8.2 `io_service::poll_one` (2 of 2 overloads)

Run the `io_service` object's event processing loop to execute one ready handler.

```
std::size_t poll_one(
    asio::error_code & ec);
```

The `poll_one()` function runs at most one handler that is ready to run, without blocking.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of handlers that were executed.

5.128.9 `io_service::post`

Request the `io_service` to invoke the given handler and return immediately.

```
template<
    typename CompletionHandler>
void-or-deduced post(
    CompletionHandler handler);
```

This function is used to ask the `io_service` to execute the given handler, but without allowing the `io_service` to call the handler from inside this function.

The `io_service` guarantees that the handler will only be called in a thread in which the `run()`, `run_one()`, `poll()` or `poll_one()` member functions is currently being invoked.

Parameters

handler The handler to be called. The `io_service` will make a copy of the handler object as required. The function signature of the handler must be:

```
void handler();
```

Remarks

This function throws an exception only if:

- the handler's `asio_handler_allocate` function; or
- the handler's copy constructor

throws an exception.

5.128.10 `io_service::reset`

Reset the `io_service` in preparation for a subsequent `run()` invocation.

```
void reset();
```

This function must be called prior to any second or later set of invocations of the `run()`, `run_one()`, `poll()` or `poll_one()` functions when a previous invocation of these functions returned due to the `io_service` being stopped or running out of work. After a call to `reset()`, the `io_service` object's `stopped()` function will return `false`.

This function must not be called while there are any unfinished calls to the `run()`, `run_one()`, `poll()` or `poll_one()` functions.

5.128.11 `io_service::run`

Run the `io_service` object's event processing loop.

```
std::size_t run();  
  
std::size_t run(  
    asio::error_code & ec);
```

5.128.11.1 `io_service::run (1 of 2 overloads)`

Run the `io_service` object's event processing loop.

```
std::size_t run();
```

The `run()` function blocks until all work has finished and there are no more handlers to be dispatched, or until the `io_service` has been stopped.

Multiple threads may call the `run()` function to set up a pool of threads from which the `io_service` may execute handlers. All threads that are waiting in the pool are equivalent and the `io_service` may choose any one of them to invoke a handler.

A normal exit from the `run()` function implies that the `io_service` object is stopped (the `stopped()` function returns `true`). Subsequent calls to `run()`, `run_one()`, `poll()` or `poll_one()` will return immediately unless there is a prior call to `reset()`.

Return Value

The number of handlers that were executed.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The `run()` function must not be called from a thread that is currently calling one of `run()`, `run_one()`, `poll()` or `poll_one()` on the same `io_service` object.

The `poll()` function may also be used to dispatch ready handlers, but without blocking.

5.128.11.2 `io_service::run (2 of 2 overloads)`

Run the `io_service` object's event processing loop.

```
std::size_t run(  
    asio::error_code & ec);
```

The `run()` function blocks until all work has finished and there are no more handlers to be dispatched, or until the `io_service` has been stopped.

Multiple threads may call the `run()` function to set up a pool of threads from which the `io_service` may execute handlers. All threads that are waiting in the pool are equivalent and the `io_service` may choose any one of them to invoke a handler.

A normal exit from the `run()` function implies that the `io_service` object is stopped (the `stopped()` function returns `true`). Subsequent calls to `run()`, `run_one()`, `poll()` or `poll_one()` will return immediately unless there is a prior call to `reset()`.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of handlers that were executed.

Remarks

The `run()` function must not be called from a thread that is currently calling one of `run()`, `run_one()`, `poll()` or `poll_one()` on the same `io_service` object.

The `poll()` function may also be used to dispatch ready handlers, but without blocking.

5.128.12 `io_service::run_one`

Run the `io_service` object's event processing loop to execute at most one handler.

```
std::size_t run_one();  
  
std::size_t run_one(  
    asio::error_code & ec);
```

5.128.12.1 `io_service::run_one (1 of 2 overloads)`

Run the `io_service` object's event processing loop to execute at most one handler.

```
std::size_t run_one();
```

The `run_one()` function blocks until one handler has been dispatched, or until the `io_service` has been stopped.

Return Value

The number of handlers that were executed. A zero return value implies that the `io_service` object is stopped (the `stopped()` function returns `true`). Subsequent calls to `run()`, `run_one()`, `poll()` or `poll_one()` will return immediately unless there is a prior call to `reset()`.

Exceptions

`asio::system_error` Thrown on failure.

5.128.12.2 `io_service::run_one (2 of 2 overloads)`

Run the `io_service` object's event processing loop to execute at most one handler.

```
std::size_t run_one(  
    asio::error_code & ec);
```

The `run_one()` function blocks until one handler has been dispatched, or until the `io_service` has been stopped.

Return Value

The number of handlers that were executed. A zero return value implies that the `io_service` object is stopped (the `stopped()` function returns `true`). Subsequent calls to `run()`, `run_one()`, `poll()` or `poll_one()` will return immediately unless there is a prior call to `reset()`.

The number of handlers that were executed.

5.128.13 `io_service::stop`

Stop the `io_service` object's event processing loop.

```
void stop();
```

This function does not block, but instead simply signals the `io_service` to stop. All invocations of its `run()` or `run_one()` member functions should return as soon as possible. Subsequent calls to `run()`, `run_one()`, `poll()` or `poll_one()` will return immediately until `reset()` is called.

5.128.14 `io_service::stopped`

Determine whether the `io_service` object has been stopped.

```
bool stopped() const;
```

This function is used to determine whether an `io_service` object has been stopped, either through an explicit call to `stop()`, or due to running out of work. When an `io_service` object is stopped, calls to `run()`, `run_one()`, `poll()` or `poll_one()` will return immediately without invoking any handlers.

Return Value

`true` if the `io_service` object is stopped, otherwise `false`.

5.128.15 `io_service::use_service`

Obtain the service object corresponding to the given type.

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

This function is used to locate a service object that corresponds to the given service type. If there is no existing implementation of the service, then the `io_service` will create a new instance of the service.

Parameters

ios The `io_service` object that owns the service.

Return Value

The service interface implementing the specified service type. Ownership of the service interface is not transferred to the caller.

Requirements

Header: asio/io_service.hpp

Convenience header: asio.hpp

5.128.16 io_service::wrap

Create a new handler that automatically dispatches the wrapped handler on the `io_service`.

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

This function is used to create a new handler function object that, when invoked, will automatically pass the wrapped handler to the `io_service` object's dispatch function.

Parameters

handler The handler to be wrapped. The `io_service` will make a copy of the handler object as required. The function signature of the handler must be:

```
void handler(A1 a1, ... An an);
```

Return Value

A function object that, when invoked, passes the wrapped handler to the `io_service` object's dispatch function. Given a function object with the signature:

```
R f(A1 a1, ... An an);
```

If this function object is passed to the wrap function like so:

```
io_service.wrap(f);
```

then the return value is a function object with the signature

```
void g(A1 a1, ... An an);
```

that, when invoked, executes code equivalent to:

```
io_service.dispatch(boost::bind(f, a1, ... an));
```

5.128.17 io_service::~io_service

Destructor.

```
~io_service();
```

On destruction, the `io_service` performs the following sequence of operations:

- For each service object `svc` in the `io_service` set, in reverse order of the beginning of service object lifetime, performs `svc->shutdown_service()`.
- Uninvoked handler objects that were scheduled for deferred invocation on the `io_service`, or any associated strand, are destroyed.
- 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)`.

Remarks

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.

5.129 io_service::id

Class used to uniquely identify a service.

```
class id :  
    noncopyable
```

Member Functions

Name	Description
id	Constructor.

Requirements

Header: `asio/io_service.hpp`

Convenience header: `asio.hpp`

5.129.1 io_service::id::id

Constructor.

```
id();
```

5.130 io_service::service

Base class for all `io_service` services.

```
class service :  
    noncopyable
```

Member Functions

Name	Description
get_io_service	Get the io_service object that owns the service.

Protected Member Functions

Name	Description
service	Constructor.
~service	Destructor.

Private Member Functions

Name	Description
fork_service	Handle notification of a fork-related event to perform any necessary housekeeping.
shutdown_service	Destroy all user-defined handler objects owned by the service.

Requirements

Header: asio/io_service.hpp

Convenience header: asio.hpp

5.130.1 io_service::service::get_io_service

Get the **io_service** object that owns the service.

```
asio::io_service & get_io_service();
```

5.130.2 io_service::service::service

Constructor.

```
service(
    asio::io_service & owner);
```

Parameters

owner The **io_service** object that owns the service.

5.130.3 `io_service::service::~service`

Destructor.

```
virtual ~service();
```

5.130.4 `io_service::service::fork_service`

Handle notification of a fork-related event to perform any necessary housekeeping.

```
virtual void fork_service(  
    asio::io_service::fork_event event);
```

This function is not a pure virtual so that services only have to implement it if necessary. The default implementation does nothing.

5.130.5 `io_service::service::shutdown_service`

Destroy all user-defined handler objects owned by the service.

```
void shutdown_service();
```

5.131 `io_service::strand`

Provides serialised handler execution.

```
class strand
```

Member Functions

Name	Description
<code>dispatch</code>	Request the strand to invoke the given handler.
<code>get_io_service</code>	Get the <code>io_service</code> associated with the strand.
<code>post</code>	Request the strand to invoke the given handler and return immediately.
<code>running_in_this_thread</code>	Determine whether the strand is running in the current thread.
<code>strand</code>	Constructor.
<code>wrap</code>	Create a new handler that automatically dispatches the wrapped handler on the strand.
<code>~strand</code>	Destructor.

The `io_service::strand` class provides the ability to post and dispatch handlers with the guarantee that none of those handlers will execute concurrently.

Order of handler invocation

Given:

- a strand object `s`
- an object `a` meeting completion handler requirements
- an object `a1` which is an arbitrary copy of `a` made by the implementation
- an object `b` meeting completion handler requirements
- an object `b1` which is an arbitrary copy of `b` made by the implementation

if any of the following conditions are true:

- `s.post(a)` happens-before `s.post(b)`
- `s.post(a)` happens-before `s.dispatch(b)`, where the latter is performed outside the strand
- `s.dispatch(a)` happens-before `s.post(b)`, where the former is performed outside the strand
- `s.dispatch(a)` happens-before `s.dispatch(b)`, where both are performed outside the strand

then `asio_handler_invoke(a1, &a1)` happens-before `asio_handler_invoke(b1, &b1)`.

Note that in the following case:

```
async_op_1(..., s.wrap(a));
async_op_2(..., s.wrap(b));
```

the completion of the first async operation will perform `s.dispatch(a)`, and the second will perform `s.dispatch(b)`, but the order in which those are performed is unspecified. That is, you cannot state whether one happens-before the other. Therefore none of the above conditions are met and no ordering guarantee is made.

Remarks

The implementation makes no guarantee that handlers posted or dispatched through different strand objects will be invoked concurrently.

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: `asio/strand.hpp`

Convenience header: `asio.hpp`

5.131.1 `io_service::strand::dispatch`

Request the strand to invoke the given handler.

```
template<
    typename CompletionHandler>
void-or-deduced dispatch(
    CompletionHandler handler);
```

This function is used to ask the strand to execute the given handler.

The strand object guarantees that handlers posted or dispatched through the strand will not be executed concurrently. The handler may be executed inside this function if the guarantee can be met. If this function is called from within a handler that was posted or dispatched through the same strand, then the new handler will be executed immediately.

The strand's guarantee is in addition to the guarantee provided by the underlying `io_service`. The `io_service` guarantees that the handler will only be called in a thread in which the `io_service`'s run member function is currently being invoked.

Parameters

handler The handler to be called. The strand will make a copy of the handler object as required. The function signature of the handler must be:

```
void handler();
```

5.131.2 `io_service::strand::get_io_service`

Get the `io_service` associated with the strand.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the `io_service` object that the strand uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_service` object that the strand will use to dispatch handlers. Ownership is not transferred to the caller.

5.131.3 `io_service::strand::post`

Request the strand to invoke the given handler and return immediately.

```
template<
    typename CompletionHandler>
void-or-deduced post(
    CompletionHandler handler);
```

This function is used to ask the strand to execute the given handler, but without allowing the strand to call the handler from inside this function.

The strand object guarantees that handlers posted or dispatched through the strand will not be executed concurrently. The strand's guarantee is in addition to the guarantee provided by the underlying `io_service`. The `io_service` guarantees that the handler will only be called in a thread in which the `io_service`'s run member function is currently being invoked.

Parameters

handler The handler to be called. The strand will make a copy of the handler object as required. The function signature of the handler must be:

```
void handler();
```

5.131.4 io_service::strand::running_in_this_thread

Determine whether the strand is running in the current thread.

```
bool running_in_this_thread() const;
```

Return Value

true if the current thread is executing a handler that was submitted to the strand using post(), dispatch() or wrap(). Otherwise returns false.

5.131.5 io_service::strand::strand

Constructor.

```
strand(  
    asio::io_service & io_service);
```

Constructs the strand.

Parameters

io_service The [io_service](#) object that the strand will use to dispatch handlers that are ready to be run.

5.131.6 io_service::strand::wrap

Create a new handler that automatically dispatches the wrapped handler on the strand.

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

This function is used to create a new handler function object that, when invoked, will automatically pass the wrapped handler to the strand's dispatch function.

Parameters

handler The handler to be wrapped. The strand will make a copy of the handler object as required. The function signature of the handler must be:

```
void handler(A1 a1, ... An an);
```

Return Value

A function object that, when invoked, passes the wrapped handler to the strand's dispatch function. Given a function object with the signature:

```
R f(A1 a1, ... An an);
```

If this function object is passed to the wrap function like so:

```
strand.wrap(f);
```

then the return value is a function object with the signature

```
void g(A1 a1, ... An an);
```

that, when invoked, executes code equivalent to:

```
strand.dispatch(boost::bind(f, a1, ... an));
```

5.131.7 io_service::strand::~strand

Destructor.

```
~strand();
```

Destroys a 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.

5.132 io_service::work

Class to inform the `io_service` when it has work to do.

```
class work
```

Member Functions

Name	Description
<code>get_io_service</code>	Get the <code>io_service</code> associated with the work.
<code>work</code>	Constructor notifies the <code>io_service</code> that work is starting. Copy constructor notifies the <code>io_service</code> that work is starting.
<code>~work</code>	Destructor notifies the <code>io_service</code> that the work is complete.

The work class is used to inform the `io_service` when work starts and finishes. This ensures that the `io_service` object's `run()` function will not exit while work is underway, and that it does exit when there is no unfinished work remaining.

The work class is copy-constructible so that it may be used as a data member in a handler class. It is not assignable.

Requirements

Header: asio/io_service.hpp

Convenience header: asio.hpp

5.132.1 io_service::work::get_io_service

Get the **io_service** associated with the work.

```
asio::io_service & get_io_service();
```

5.132.2 io_service::work::work

Constructor notifies the **io_service** that work is starting.

```
explicit work(
    asio::io_service & io_service);
```

Copy constructor notifies the **io_service** that work is starting.

```
work(
    const work & other);
```

5.132.2.1 io_service::work::work (1 of 2 overloads)

Constructor notifies the **io_service** that work is starting.

```
work(
    asio::io_service & io_service);
```

The constructor is used to inform the **io_service** that some work has begun. This ensures that the **io_service** object's `run()` function will not exit while the work is underway.

5.132.2.2 io_service::work::work (2 of 2 overloads)

Copy constructor notifies the **io_service** that work is starting.

```
work(
    const work & other);
```

The constructor is used to inform the **io_service** that some work has begun. This ensures that the **io_service** object's `run()` function will not exit while the work is underway.

5.132.3 io_service::work::~work

Destructor notifies the **io_service** that the work is complete.

```
~work();
```

The destructor is used to inform the **io_service** that some work has finished. Once the count of unfinished work reaches zero, the **io_service** object's `run()` function is permitted to exit.

5.133 ip::address

Implements version-independent IP addresses.

```
class address
```

Member Functions

Name	Description
address	Default constructor. Construct an address from an IPv4 address. Construct an address from an IPv6 address. Copy constructor.
from_string	Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.
is_loopback	Determine whether the address is a loopback address.
is_multicast	Determine whether the address is a multicast address.
is_unspecified	Determine whether the address is unspecified.
is_v4	Get whether the address is an IP version 4 address.
is_v6	Get whether the address is an IP version 6 address.
operator=	Assign from another address. Assign from an IPv4 address. Assign from an IPv6 address.
to_string	Get the address as a string in dotted decimal format.
to_v4	Get the address as an IP version 4 address.
to_v6	Get the address as an IP version 6 address.

Friends

Name	Description
operator!=	Compare two addresses for inequality.
operator<	Compare addresses for ordering.
operator<=	Compare addresses for ordering.
operator==	Compare two addresses for equality.
operator>	Compare addresses for ordering.

Name	Description
operator>=	Compare addresses for ordering.

Related Functions

Name	Description
operator<<	Output an address as a string.

The `ip::address` class provides the ability to use either IP version 4 or version 6 addresses.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/address.hpp`

Convenience header: `asio.hpp`

5.133.1 ip::address::address

Default constructor.

```
address();
```

Construct an address from an IPv4 address.

```
address(
    const asio::ip::address_v4 & ipv4_address);
```

Construct an address from an IPv6 address.

```
address(
    const asio::ip::address_v6 & ipv6_address);
```

Copy constructor.

```
address(
    const address & other);
```

5.133.1.1 ip::address::address (1 of 4 overloads)

Default constructor.

```
address();
```

5.133.1.2 ip::address::address (2 of 4 overloads)

Construct an address from an IPv4 address.

```
address(  
    const asio::ip::address_v4 & ipv4_address);
```

5.133.1.3 ip::address::address (3 of 4 overloads)

Construct an address from an IPv6 address.

```
address(  
    const asio::ip::address_v6 & ipv6_address);
```

5.133.1.4 ip::address::address (4 of 4 overloads)

Copy constructor.

```
address(  
    const address & other);
```

5.133.2 ip::address::from_string

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
static address from_string(  
    const char * str);  
  
static address from_string(  
    const char * str,  
    asio::error_code & ec);  
  
static address from_string(  
    const std::string & str);  
  
static address from_string(  
    const std::string & str,  
    asio::error_code & ec);
```

5.133.2.1 ip::address::from_string (1 of 4 overloads)

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
static address from_string(  
    const char * str);
```

5.133.2.2 ip::address::from_string (2 of 4 overloads)

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
static address from_string(  
    const char * str,  
    asio::error_code & ec);
```

5.133.2.3 ip::address::from_string (3 of 4 overloads)

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
static address from_string(
    const std::string & str);
```

5.133.2.4 ip::address::from_string (4 of 4 overloads)

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
static address from_string(
    const std::string & str,
    asio::error_code & ec);
```

5.133.3 ip::address::is_loopback

Determine whether the address is a loopback address.

```
bool is_loopback() const;
```

5.133.4 ip::address::is_multicast

Determine whether the address is a multicast address.

```
bool is_multicast() const;
```

5.133.5 ip::address::is_unspecified

Determine whether the address is unspecified.

```
bool is_unspecified() const;
```

5.133.6 ip::address::is_v4

Get whether the address is an IP version 4 address.

```
bool is_v4() const;
```

5.133.7 ip::address::is_v6

Get whether the address is an IP version 6 address.

```
bool is_v6() const;
```

5.133.8 ip::address::operator!=

Compare two addresses for inequality.

```
friend bool operator!=(
    const address & a1,
    const address & a2);
```

Requirements

Header: asio/ip/address.hpp

Convenience header: asio.hpp

5.133.9 ip::address::operator<

Compare addresses for ordering.

```
friend bool operator<(
    const address & a1,
    const address & a2);
```

Requirements

Header: asio/ip/address.hpp

Convenience header: asio.hpp

5.133.10 ip::address::operator<<

Output an address as a string.

```
template<
    typename Elem,
    typename Traits>
std::basic_ostream< Elem, Traits > & operator<<(
    std::basic_ostream< Elem, Traits > & os,
    const address & addr);
```

Used to output a human-readable string for a specified address.

Parameters

os The output stream to which the string will be written.

addr The address to be written.

Return Value

The output stream.

5.133.11 ip::address::operator<=

Compare addresses for ordering.

```
friend bool operator<=
    const address & a1,
    const address & a2);
```

Requirements

Header: asio/ip/address.hpp

Convenience header: asio.hpp

5.133.12 ip::address::operator=

Assign from another address.

```
address & operator=(  
    const address & other);
```

Assign from an IPv4 address.

```
address & operator=(  
    const asio::ip::address_v4 & ipv4_address);
```

Assign from an IPv6 address.

```
address & operator=(  
    const asio::ip::address_v6 & ipv6_address);
```

5.133.12.1 ip::address::operator= (1 of 3 overloads)

Assign from another address.

```
address & operator=(  
    const address & other);
```

5.133.12.2 ip::address::operator= (2 of 3 overloads)

Assign from an IPv4 address.

```
address & operator=(  
    const asio::ip::address_v4 & ipv4_address);
```

5.133.12.3 ip::address::operator= (3 of 3 overloads)

Assign from an IPv6 address.

```
address & operator=(  
    const asio::ip::address_v6 & ipv6_address);
```

5.133.13 ip::address::operator==

Compare two addresses for equality.

```
friend bool operator==(  
    const address & a1,  
    const address & a2);
```

Requirements

Header: asio/ip/address.hpp

Convenience header: asio.hpp

5.133.14 ip::address::operator>

Compare addresses for ordering.

```
friend bool operator>(
    const address & a1,
    const address & a2);
```

Requirements

Header: asio/ip/address.hpp

Convenience header: asio.hpp

5.133.15 ip::address::operator>=

Compare addresses for ordering.

```
friend bool operator>=(
    const address & a1,
    const address & a2);
```

Requirements

Header: asio/ip/address.hpp

Convenience header: asio.hpp

5.133.16 ip::address::to_string

Get the address as a string in dotted decimal format.

```
std::string to_string() const;

std::string to_string(
    asio::error_code & ec) const;
```

5.133.16.1 ip::address::to_string (1 of 2 overloads)

Get the address as a string in dotted decimal format.

```
std::string to_string() const;
```

5.133.16.2 ip::address::to_string (2 of 2 overloads)

Get the address as a string in dotted decimal format.

```
std::string to_string(
    asio::error_code & ec) const;
```

5.133.17 ip::address::to_v4

Get the address as an IP version 4 address.

```
asio::ip::address_v4 to_v4() const;
```

5.133.18 ip::address::to_v6

Get the address as an IP version 6 address.

```
asio::ip::address_v6 to_v6() const;
```

5.134 ip::address_v4

Implements IP version 4 style addresses.

```
class address_v4
```

Types

Name	Description
bytes_type	The type used to represent an address as an array of bytes.

Member Functions

Name	Description
address_v4	Default constructor. Construct an address from raw bytes. Construct an address from a unsigned long in host byte order. Copy constructor.
any	Obtain an address object that represents any address.
broadcast	Obtain an address object that represents the broadcast address. Obtain an address object that represents the broadcast address that corresponds to the specified address and netmask.
from_string	Create an address from an IP address string in dotted decimal form.
is_class_a	Determine whether the address is a class A address.
is_class_b	Determine whether the address is a class B address.
is_class_c	Determine whether the address is a class C address.
is_loopback	Determine whether the address is a loopback address.

Name	Description
is_multicast	Determine whether the address is a multicast address.
is_unspecified	Determine whether the address is unspecified.
loopback	Obtain an address object that represents the loopback address.
netmask	Obtain the netmask that corresponds to the address, based on its address class.
operator=	Assign from another address.
to_bytes	Get the address in bytes, in network byte order.
to_string	Get the address as a string in dotted decimal format.
to_ulong	Get the address as an unsigned long in host byte order.

Friends

Name	Description
operator!=	Compare two addresses for inequality.
operator<	Compare addresses for ordering.
operator<=	Compare addresses for ordering.
operator==	Compare two addresses for equality.
operator>	Compare addresses for ordering.
operator>=	Compare addresses for ordering.

Related Functions

Name	Description
operator<<	Output an address as a string.

The `ip::address_v4` class provides the ability to use and manipulate IP version 4 addresses.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v4.hpp

Convenience header: asio.hpp

5.134.1 ip::address_v4::address_v4

Default constructor.

```
address_v4();
```

Construct an address from raw bytes.

```
explicit address_v4(
    const bytes_type & bytes);
```

Construct an address from a unsigned long in host byte order.

```
explicit address_v4(
    unsigned long addr);
```

Copy constructor.

```
address_v4(
    const address_v4 & other);
```

5.134.1.1 ip::address_v4::address_v4 (1 of 4 overloads)

Default constructor.

```
address_v4();
```

5.134.1.2 ip::address_v4::address_v4 (2 of 4 overloads)

Construct an address from raw bytes.

```
address_v4(
    const bytes_type & bytes);
```

5.134.1.3 ip::address_v4::address_v4 (3 of 4 overloads)

Construct an address from a unsigned long in host byte order.

```
address_v4(
    unsigned long addr);
```

5.134.1.4 ip::address_v4::address_v4 (4 of 4 overloads)

Copy constructor.

```
address_v4(
    const address_v4 & other);
```

5.134.2 ip::address_v4::any

Obtain an address object that represents any address.

```
static address_v4 any();
```

5.134.3 ip::address_v4::broadcast

Obtain an address object that represents the broadcast address.

```
static address_v4 broadcast();
```

Obtain an address object that represents the broadcast address that corresponds to the specified address and netmask.

```
static address_v4 broadcast(
    const address_v4 & addr,
    const address_v4 & mask);
```

5.134.3.1 ip::address_v4::broadcast (1 of 2 overloads)

Obtain an address object that represents the broadcast address.

```
static address_v4 broadcast();
```

5.134.3.2 ip::address_v4::broadcast (2 of 2 overloads)

Obtain an address object that represents the broadcast address that corresponds to the specified address and netmask.

```
static address_v4 broadcast(
    const address_v4 & addr,
    const address_v4 & mask);
```

5.134.4 ip::address_v4::bytes_type

The type used to represent an address as an array of bytes.

```
typedef array< unsigned char, 4 > bytes_type;
```

Remarks

This type is defined in terms of the C++0x template `std::array` when it is available. Otherwise, it uses `boost::array`.

Requirements

Header: `asio/ip/address_v4.hpp`

Convenience header: `asio.hpp`

5.134.5 ip::address_v4::from_string

Create an address from an IP address string in dotted decimal form.

```
static address_v4 from_string(
    const char * str);

static address_v4 from_string(
    const char * str,
    asio::error_code & ec);

static address_v4 from_string(
    const std::string & str);

static address_v4 from_string(
    const std::string & str,
    asio::error_code & ec);
```

5.134.5.1 ip::address_v4::from_string (1 of 4 overloads)

Create an address from an IP address string in dotted decimal form.

```
static address_v4 from_string(
    const char * str);
```

5.134.5.2 ip::address_v4::from_string (2 of 4 overloads)

Create an address from an IP address string in dotted decimal form.

```
static address_v4 from_string(
    const char * str,
    asio::error_code & ec);
```

5.134.5.3 ip::address_v4::from_string (3 of 4 overloads)

Create an address from an IP address string in dotted decimal form.

```
static address_v4 from_string(
    const std::string & str);
```

5.134.5.4 ip::address_v4::from_string (4 of 4 overloads)

Create an address from an IP address string in dotted decimal form.

```
static address_v4 from_string(
    const std::string & str,
    asio::error_code & ec);
```

5.134.6 ip::address_v4::is_class_a

Determine whether the address is a class A address.

```
bool is_class_a() const;
```

5.134.7 ip::address_v4::is_class_b

Determine whether the address is a class B address.

```
bool is_class_b() const;
```

5.134.8 ip::address_v4::is_class_c

Determine whether the address is a class C address.

```
bool is_class_c() const;
```

5.134.9 ip::address_v4::is_loopback

Determine whether the address is a loopback address.

```
bool is_loopback() const;
```

5.134.10 ip::address_v4::is_multicast

Determine whether the address is a multicast address.

```
bool is_multicast() const;
```

5.134.11 ip::address_v4::is_unspecified

Determine whether the address is unspecified.

```
bool is_unspecified() const;
```

5.134.12 ip::address_v4::loopback

Obtain an address object that represents the loopback address.

```
static address_v4 loopback();
```

5.134.13 ip::address_v4::netmask

Obtain the netmask that corresponds to the address, based on its address class.

```
static address_v4 netmask(
    const address_v4 & addr);
```

5.134.14 ip::address_v4::operator!=

Compare two addresses for inequality.

```
friend bool operator!=(
    const address_v4 & a1,
    const address_v4 & a2);
```

Requirements

Header: asio/ip/address_v4.hpp

Convenience header: asio.hpp

5.134.15 ip::address_v4::operator<

Compare addresses for ordering.

```
friend bool operator<(
    const address_v4 & a1,
    const address_v4 & a2);
```

Requirements

Header: asio/ip/address_v4.hpp

Convenience header: asio.hpp

5.134.16 ip::address_v4::operator<<

Output an address as a string.

```
template<
    typename Elem,
    typename Traits>
std::basic_ostream< Elem, Traits > & operator<<(
    std::basic_ostream< Elem, Traits > & os,
    const address_v4 & addr);
```

Used to output a human-readable string for a specified address.

Parameters

os The output stream to which the string will be written.

addr The address to be written.

Return Value

The output stream.

5.134.17 ip::address_v4::operator<=

Compare addresses for ordering.

```
friend bool operator<=
    const address_v4 & a1,
    const address_v4 & a2);
```

Requirements

Header: asio/ip/address_v4.hpp

Convenience header: asio.hpp

5.134.18 ip::address_v4::operator=

Assign from another address.

```
address_v4 & operator=(  
    const address_v4 & other);
```

5.134.19 ip::address_v4::operator==

Compare two addresses for equality.

```
friend bool operator==(  
    const address_v4 & a1,  
    const address_v4 & a2);
```

Requirements

Header: asio/ip/address_v4.hpp

Convenience header: asio.hpp

5.134.20 ip::address_v4::operator>

Compare addresses for ordering.

```
friend bool operator>(  
    const address_v4 & a1,  
    const address_v4 & a2);
```

Requirements

Header: asio/ip/address_v4.hpp

Convenience header: asio.hpp

5.134.21 ip::address_v4::operator>=

Compare addresses for ordering.

```
friend bool operator>=(  
    const address_v4 & a1,  
    const address_v4 & a2);
```

Requirements

Header: asio/ip/address_v4.hpp

Convenience header: asio.hpp

5.134.22 ip::address_v4::to_bytes

Get the address in bytes, in network byte order.

```
bytes_type to_bytes() const;
```

5.134.23 ip::address_v4::to_string

Get the address as a string in dotted decimal format.

```
std::string to_string() const;  
  
std::string to_string(  
    asio::error_code & ec) const;
```

5.134.23.1 ip::address_v4::to_string (1 of 2 overloads)

Get the address as a string in dotted decimal format.

```
std::string to_string() const;
```

5.134.23.2 ip::address_v4::to_string (2 of 2 overloads)

Get the address as a string in dotted decimal format.

```
std::string to_string(  
    asio::error_code & ec) const;
```

5.134.24 ip::address_v4::to_ulong

Get the address as an unsigned long in host byte order.

```
unsigned long to_ulong() const;
```

5.135 ip::address_v6

Implements IP version 6 style addresses.

```
class address_v6
```

Types

Name	Description
bytes_type	The type used to represent an address as an array of bytes.

Member Functions

Name	Description
address_v6	Default constructor. Construct an address from raw bytes and scope ID. Copy constructor.

Name	Description
any	Obtain an address object that represents any address.
from_string	Create an address from an IP address string.
is_link_local	Determine whether the address is link local.
is_loopback	Determine whether the address is a loopback address.
is_multicast	Determine whether the address is a multicast address.
is_multicast_global	Determine whether the address is a global multicast address.
is_multicast_link_local	Determine whether the address is a link-local multicast address.
is_multicast_node_local	Determine whether the address is a node-local multicast address.
is_multicast_org_local	Determine whether the address is a org-local multicast address.
is_multicast_site_local	Determine whether the address is a site-local multicast address.
is_site_local	Determine whether the address is site local.
is_unspecified	Determine whether the address is unspecified.
is_v4_compatible	Determine whether the address is an IPv4-compatible address.
is_v4_mapped	Determine whether the address is a mapped IPv4 address.
loopback	Obtain an address object that represents the loopback address.
operator=	Assign from another address.
scope_id	The scope ID of the address.
to_bytes	Get the address in bytes, in network byte order.
to_string	Get the address as a string.
to_v4	Converts an IPv4-mapped or IPv4-compatible address to an IPv4 address.
v4_compatible	Create an IPv4-compatible IPv6 address.
v4_mapped	Create an IPv4-mapped IPv6 address.

Friends

Name	Description
operator!=	Compare two addresses for inequality.
operator<	Compare addresses for ordering.
operator<=	Compare addresses for ordering.
operator==	Compare two addresses for equality.
operator>	Compare addresses for ordering.
operator>=	Compare addresses for ordering.

Related Functions

Name	Description
operator<<	Output an address as a string.

The `ip::address_v6` class provides the ability to use and manipulate IP version 6 addresses.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/address_v6.hpp`

Convenience header: `asio.hpp`

5.135.1 ip::address_v6::address_v6

Default constructor.

```
address_v6();
```

Construct an address from raw bytes and scope ID.

```
explicit address_v6(
    const bytes_type & bytes,
    unsigned long scope_id = 0);
```

Copy constructor.

```
address_v6(
    const address_v6 & other);
```

5.135.1.1 ip::address_v6::address_v6 (1 of 3 overloads)

Default constructor.

```
address_v6();
```

5.135.1.2 ip::address_v6::address_v6 (2 of 3 overloads)

Construct an address from raw bytes and scope ID.

```
address_v6(  
    const bytes_type & bytes,  
    unsigned long scope_id = 0);
```

5.135.1.3 ip::address_v6::address_v6 (3 of 3 overloads)

Copy constructor.

```
address_v6(  
    const address_v6 & other);
```

5.135.2 ip::address_v6::any

Obtain an address object that represents any address.

```
static address_v6 any();
```

5.135.3 ip::address_v6::bytes_type

The type used to represent an address as an array of bytes.

```
typedef array< unsigned char, 16 > bytes_type;
```

Remarks

This type is defined in terms of the C++0x template `std::array` when it is available. Otherwise, it uses `boost::array`.

Requirements

Header: `asio/ip/address_v6.hpp`

Convenience header: `asio.hpp`

5.135.4 ip::address_v6::from_string

Create an address from an IP address string.

```
static address_v6 from_string(  
    const char * str);
```

```
static address_v6 from_string(  
    const char * str,  
    asio::error_code & ec);
```

```
static address_v6 from_string(
    const std::string & str);

static address_v6 from_string(
    const std::string & str,
    asio::error_code & ec);
```

5.135.4.1 ip::address_v6::from_string (1 of 4 overloads)

Create an address from an IP address string.

```
static address_v6 from_string(
    const char * str);
```

5.135.4.2 ip::address_v6::from_string (2 of 4 overloads)

Create an address from an IP address string.

```
static address_v6 from_string(
    const char * str,
    asio::error_code & ec);
```

5.135.4.3 ip::address_v6::from_string (3 of 4 overloads)

Create an address from an IP address string.

```
static address_v6 from_string(
    const std::string & str);
```

5.135.4.4 ip::address_v6::from_string (4 of 4 overloads)

Create an address from an IP address string.

```
static address_v6 from_string(
    const std::string & str,
    asio::error_code & ec);
```

5.135.5 ip::address_v6::is_link_local

Determine whether the address is link local.

```
bool is_link_local() const;
```

5.135.6 ip::address_v6::is_loopback

Determine whether the address is a loopback address.

```
bool is_loopback() const;
```

5.135.7 ip::address_v6::is_multicast

Determine whether the address is a multicast address.

```
bool is_multicast() const;
```

5.135.8 ip::address_v6::is_multicast_global

Determine whether the address is a global multicast address.

```
bool is_multicast_global() const;
```

5.135.9 ip::address_v6::is_multicast_link_local

Determine whether the address is a link-local multicast address.

```
bool is_multicast_link_local() const;
```

5.135.10 ip::address_v6::is_multicast_node_local

Determine whether the address is a node-local multicast address.

```
bool is_multicast_node_local() const;
```

5.135.11 ip::address_v6::is_multicast_org_local

Determine whether the address is a org-local multicast address.

```
bool is_multicast_org_local() const;
```

5.135.12 ip::address_v6::is_multicast_site_local

Determine whether the address is a site-local multicast address.

```
bool is_multicast_site_local() const;
```

5.135.13 ip::address_v6::is_site_local

Determine whether the address is site local.

```
bool is_site_local() const;
```

5.135.14 ip::address_v6::is_unspecified

Determine whether the address is unspecified.

```
bool is_unspecified() const;
```

5.135.15 ip::address_v6::is_v4_compatible

Determine whether the address is an IPv4-compatible address.

```
bool is_v4_compatible() const;
```

5.135.16 ip::address_v6::is_v4_mapped

Determine whether the address is a mapped IPv4 address.

```
bool is_v4_mapped() const;
```

5.135.17 ip::address_v6::loopback

Obtain an address object that represents the loopback address.

```
static address_v6 loopback();
```

5.135.18 ip::address_v6::operator!=

Compare two addresses for inequality.

```
friend bool operator!=(
    const address_v6 & a1,
    const address_v6 & a2);
```

Requirements

Header: asio/ip/address_v6.hpp

Convenience header: asio.hpp

5.135.19 ip::address_v6::operator<

Compare addresses for ordering.

```
friend bool operator<(
    const address_v6 & a1,
    const address_v6 & a2);
```

Requirements

Header: asio/ip/address_v6.hpp

Convenience header: asio.hpp

5.135.20 ip::address_v6::operator<<

Output an address as a string.

```
template<
    typename Elem,
    typename Traits>
std::basic_ostream< Elem, Traits > & operator<< (
    std::basic_ostream< Elem, Traits > & os,
    const address_v6 & addr);
```

Used to output a human-readable string for a specified address.

Parameters

os The output stream to which the string will be written.

addr The address to be written.

Return Value

The output stream.

5.135.21 ip::address_v6::operator<=

Compare addresses for ordering.

```
friend bool operator<=
    const address_v6 & a1,
    const address_v6 & a2);
```

Requirements

Header: asio/ip/address_v6.hpp

Convenience header: asio.hpp

5.135.22 ip::address_v6::operator=

Assign from another address.

```
address_v6 & operator=(
    const address_v6 & other);
```

5.135.23 ip::address_v6::operator==

Compare two addresses for equality.

```
friend bool operator==
    const address_v6 & a1,
    const address_v6 & a2);
```

Requirements

Header: asio/ip/address_v6.hpp

Convenience header: asio.hpp

5.135.24 ip::address_v6::operator>

Compare addresses for ordering.

```
friend bool operator>(
    const address_v6 & a1,
    const address_v6 & a2);
```

Requirements

Header: asio/ip/address_v6.hpp

Convenience header: asio.hpp

5.135.25 ip::address_v6::operator>=

Compare addresses for ordering.

```
friend bool operator>=
    const address_v6 & a1,
    const address_v6 & a2);
```

Requirements

Header: asio/ip/address_v6.hpp

Convenience header: asio.hpp

5.135.26 ip::address_v6::scope_id

The scope ID of the address.

```
unsigned long scope_id() const;

void scope_id(
    unsigned long id);
```

5.135.26.1 ip::address_v6::scope_id (1 of 2 overloads)

The scope ID of the address.

```
unsigned long scope_id() const;
```

Returns the scope ID associated with the IPv6 address.

5.135.26.2 ip::address_v6::scope_id (2 of 2 overloads)

The scope ID of the address.

```
void scope_id(
    unsigned long id);
```

Modifies the scope ID associated with the IPv6 address.

5.135.27 ip::address_v6::to_bytes

Get the address in bytes, in network byte order.

```
bytes_type to_bytes() const;
```

5.135.28 ip::address_v6::to_string

Get the address as a string.

```
std::string to_string() const;  
  
std::string to_string(  
    asio::error_code & ec) const;
```

5.135.28.1 ip::address_v6::to_string (1 of 2 overloads)

Get the address as a string.

```
std::string to_string() const;
```

5.135.28.2 ip::address_v6::to_string (2 of 2 overloads)

Get the address as a string.

```
std::string to_string(  
    asio::error_code & ec) const;
```

5.135.29 ip::address_v6::to_v4

Converts an IPv4-mapped or IPv4-compatible address to an IPv4 address.

```
address_v4 to_v4() const;
```

5.135.30 ip::address_v6::v4_compatible

Create an IPv4-compatible IPv6 address.

```
static address_v6 v4_compatible(  
    const address_v4 & addr);
```

5.135.31 ip::address_v6::v4_mapped

Create an IPv4-mapped IPv6 address.

```
static address_v6 v4_mapped(  
    const address_v4 & addr);
```

5.136 ip::basic_endpoint

Describes an endpoint for a version-independent IP socket.

```
template<  
    typename InternetProtocol>  
class basic_endpoint
```

Types

Name	Description
data_type	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
protocol_type	The protocol type associated with the endpoint.

Member Functions

Name	Description
address	Get the IP address associated with the endpoint. Set the IP address associated with the endpoint.
basic_endpoint	Default constructor. Construct an endpoint using a port number, specified in the host's byte order. The IP address will be the any address (i.e. INADDR_ANY or in6addr_any). This constructor would typically be used for accepting new connections. Construct an endpoint using a port number and an IP address. This constructor may be used for accepting connections on a specific interface or for making a connection to a remote endpoint. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
port	Get the port associated with the endpoint. The port number is always in the host's byte order. Set the port associated with the endpoint. The port number is always in the host's byte order.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.

Name	Description
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

Related Functions

Name	Description
operator<<	Output an endpoint as a string.

The `ip::basic_endpoint` class template describes an endpoint that may be associated with a particular socket.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/basic_endpoint.hpp`

Convenience header: `asio.hpp`

5.136.1 ip::basic_endpoint::address

Get the IP address associated with the endpoint.

```
asio::ip::address address() const;
```

Set the IP address associated with the endpoint.

```
void address(
    const asio::ip::address & addr);
```

5.136.1.1 ip::basic_endpoint::address (1 of 2 overloads)

Get the IP address associated with the endpoint.

```
asio::ip::address address() const;
```

5.136.1.2 ip::basic_endpoint::address (2 of 2 overloads)

Set the IP address associated with the endpoint.

```
void address(
    const asio::ip::address & addr);
```

5.136.2 ip::basic_endpoint::basic_endpoint

Default constructor.

```
basic_endpoint();
```

Construct an endpoint using a port number, specified in the host's byte order. The IP address will be the any address (i.e. IN-ADDR_ANY or in6addr_any). This constructor would typically be used for accepting new connections.

```
basic_endpoint(
    const InternetProtocol & internet_protocol,
    unsigned short port_num);
```

Construct an endpoint using a port number and an IP address. This constructor may be used for accepting connections on a specific interface or for making a connection to a remote endpoint.

```
basic_endpoint(
    const asio::ip::address & addr,
    unsigned short port_num);
```

Copy constructor.

```
basic_endpoint(
    const basic_endpoint & other);
```

5.136.2.1 ip::basic_endpoint::basic_endpoint (1 of 4 overloads)

Default constructor.

```
basic_endpoint();
```

5.136.2.2 ip::basic_endpoint::basic_endpoint (2 of 4 overloads)

Construct an endpoint using a port number, specified in the host's byte order. The IP address will be the any address (i.e. IN-ADDR_ANY or in6addr_any). This constructor would typically be used for accepting new connections.

```
basic_endpoint(
    const InternetProtocol & internet_protocol,
    unsigned short port_num);
```

Examples

To initialise an IPv4 TCP endpoint for port 1234, use:

```
asio::ip::tcp::endpoint ep(asio::ip::tcp::v4(), 1234);
```

To specify an IPv6 UDP endpoint for port 9876, use:

```
asio::ip::udp::endpoint ep(asio::ip::udp::v6(), 9876);
```

5.136.2.3 ip::basic_endpoint::basic_endpoint (3 of 4 overloads)

Construct an endpoint using a port number and an IP address. This constructor may be used for accepting connections on a specific interface or for making a connection to a remote endpoint.

```
basic_endpoint(
    const asio::ip::address & addr,
    unsigned short port_num);
```

5.136.2.4 ip::basic_endpoint::basic_endpoint (4 of 4 overloads)

Copy constructor.

```
basic_endpoint(
    const basic_endpoint & other);
```

5.136.3 ip::basic_endpoint::capacity

Get the capacity of the endpoint in the native type.

```
std::size_t capacity() const;
```

5.136.4 ip::basic_endpoint::data

Get the underlying endpoint in the native type.

```
data_type * data();
```

```
const data_type * data() const;
```

5.136.4.1 ip::basic_endpoint::data (1 of 2 overloads)

Get the underlying endpoint in the native type.

```
data_type * data();
```

5.136.4.2 ip::basic_endpoint::data (2 of 2 overloads)

Get the underlying endpoint in the native type.

```
const data_type * data() const;
```

5.136.5 ip::basic_endpoint::data_type

The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.

```
typedef implementation_defined data_type;
```

Requirements

Header: asio/ip/basic_endpoint.hpp

Convenience header: asio.hpp

5.136.6 ip::basic_endpoint::operator!=

Compare two endpoints for inequality.

```
friend bool operator!=(
    const basic_endpoint< InternetProtocol > & e1,
    const basic_endpoint< InternetProtocol > & e2);
```

Requirements

Header: asio/ip/basic_endpoint.hpp

Convenience header: asio.hpp

5.136.7 ip::basic_endpoint::operator<

Compare endpoints for ordering.

```
friend bool operator<(
    const basic_endpoint< InternetProtocol > & e1,
    const basic_endpoint< InternetProtocol > & e2);
```

Requirements

Header: asio/ip/basic_endpoint.hpp

Convenience header: asio.hpp

5.136.8 ip::basic_endpoint::operator<<

Output an endpoint as a string.

```
std::basic_ostream< Elem, Traits > & operator<<(
    std::basic_ostream< Elem, Traits > & os,
    const basic_endpoint< InternetProtocol > & endpoint);
```

Used to output a human-readable string for a specified endpoint.

Parameters

os The output stream to which the string will be written.

endpoint The endpoint to be written.

Return Value

The output stream.

5.136.9 ip::basic_endpoint::operator<=

Compare endpoints for ordering.

```
friend bool operator<=(
    const basic_endpoint< InternetProtocol > & e1,
    const basic_endpoint< InternetProtocol > & e2);
```

Requirements

Header: asio/ip/basic_endpoint.hpp

Convenience header: asio.hpp

5.136.10 ip::basic_endpoint::operator=

Assign from another endpoint.

```
basic_endpoint & operator=(  
    const basic_endpoint & other);
```

5.136.11 ip::basic_endpoint::operator==

Compare two endpoints for equality.

```
friend bool operator==(  
    const basic_endpoint< InternetProtocol > & e1,  
    const basic_endpoint< InternetProtocol > & e2);
```

Requirements

Header: asio/ip/basic_endpoint.hpp

Convenience header: asio.hpp

5.136.12 ip::basic_endpoint::operator>

Compare endpoints for ordering.

```
friend bool operator>(  
    const basic_endpoint< InternetProtocol > & e1,  
    const basic_endpoint< InternetProtocol > & e2);
```

Requirements

Header: asio/ip/basic_endpoint.hpp

Convenience header: asio.hpp

5.136.13 ip::basic_endpoint::operator>=

Compare endpoints for ordering.

```
friend bool operator>=(  
    const basic_endpoint< InternetProtocol > & e1,  
    const basic_endpoint< InternetProtocol > & e2);
```

Requirements

Header: asio/ip/basic_endpoint.hpp

Convenience header: asio.hpp

5.136.14 ip::basic_endpoint::port

Get the port associated with the endpoint. The port number is always in the host's byte order.

```
unsigned short port() const;
```

Set the port associated with the endpoint. The port number is always in the host's byte order.

```
void port(
    unsigned short port_num);
```

5.136.14.1 ip::basic_endpoint::port (1 of 2 overloads)

Get the port associated with the endpoint. The port number is always in the host's byte order.

```
unsigned short port() const;
```

5.136.14.2 ip::basic_endpoint::port (2 of 2 overloads)

Set the port associated with the endpoint. The port number is always in the host's byte order.

```
void port(
    unsigned short port_num);
```

5.136.15 ip::basic_endpoint::protocol

The protocol associated with the endpoint.

```
protocol_type protocol() const;
```

5.136.16 ip::basic_endpoint::protocol_type

The protocol type associated with the endpoint.

```
typedef InternetProtocol protocol_type;
```

Requirements

Header: asio/ip/basic_endpoint.hpp

Convenience header: asio.hpp

5.136.17 ip::basic_endpoint::resize

Set the underlying size of the endpoint in the native type.

```
void resize(
    std::size_t new_size);
```

5.136.18 ip::basic_endpoint::size

Get the underlying size of the endpoint in the native type.

```
std::size_t size() const;
```

5.137 ip::basic_resolver

Provides endpoint resolution functionality.

```
template<
    typename InternetProtocol,
    typename ResolverService = resolver_service<InternetProtocol>>
class basic_resolver :
    public basic_io_object<ResolverService>
```

Types

Name	Description
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
iterator	The iterator type.
protocol_type	The protocol type.
query	The query type.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
async_resolve	Asynchronously perform forward resolution of a query to a list of entries. Asynchronously perform reverse resolution of an endpoint to a list of entries.
basic_resolver	Constructor.
cancel	Cancel any asynchronous operations that are waiting on the resolver.
get_io_service	Get the io_service associated with the object.
resolve	Perform forward resolution of a query to a list of entries. Perform reverse resolution of an endpoint to a list of entries.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `ip::basic_resolver` class template provides the ability to resolve a query to a list of endpoints.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/basic_resolver.hpp`

Convenience header: `asio.hpp`

5.137.1 `ip::basic_resolver::async_resolve`

Asynchronously perform forward resolution of a query to a list of entries.

```
template<
    typename ResolveHandler>
void-or-deduced async_resolve(
    const query & q,
    ResolveHandler handler);
```

Asynchronously perform reverse resolution of an endpoint to a list of entries.

```
template<
    typename ResolveHandler>
void-or-deduced async_resolve(
    const endpoint_type & e,
    ResolveHandler handler);
```

5.137.1.1 ip::basic_resolver::async_resolve (1 of 2 overloads)

Asynchronously perform forward resolution of a query to a list of entries.

```
template<
    typename ResolveHandler>
void-or-deduced async_resolve(
    const query & q,
    ResolveHandler handler);
```

This function is used to asynchronously resolve a query into a list of endpoint entries.

Parameters

q A query object that determines what endpoints will be returned.

handler The handler to be called when the resolve operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    resolver::iterator iterator      // Forward-only iterator that can
                                    // be used to traverse the list
                                    // of endpoint entries.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

A default constructed iterator represents the end of the list.

A successful resolve operation is guaranteed to pass at least one entry to the handler.

5.137.1.2 ip::basic_resolver::async_resolve (2 of 2 overloads)

Asynchronously perform reverse resolution of an endpoint to a list of entries.

```
template<
    typename ResolveHandler>
void-or-deduced async_resolve(
    const endpoint_type & e,
    ResolveHandler handler);
```

This function is used to asynchronously resolve an endpoint into a list of endpoint entries.

Parameters

e An endpoint object that determines what endpoints will be returned.

handler The handler to be called when the resolve operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    resolver::iterator iterator      // Forward-only iterator that can
                                    // be used to traverse the list
                                    // of endpoint entries.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

A default constructed iterator represents the end of the list.

A successful resolve operation is guaranteed to pass at least one entry to the handler.

5.137.2 ip::basic_resolver::basic_resolver

Constructor.

```
basic_resolver(  
   asio::io_service & io_service);
```

This constructor creates a [ip::basic_resolver](#).

Parameters

io_service The [io_service](#) object that the resolver will use to dispatch handlers for any asynchronous operations performed on the timer.

5.137.3 ip::basic_resolver::cancel

Cancel any asynchronous operations that are waiting on the resolver.

```
void cancel();
```

This function forces the completion of any pending asynchronous operations on the host resolver. The handler for each cancelled operation will be invoked with the [asio::error::operation_aborted](#) error code.

5.137.4 ip::basic_resolver::endpoint_type

The endpoint type.

```
typedef InternetProtocol::endpoint endpoint_type;
```

Requirements

Header: `asio/ip/basic_resolver.hpp`

Convenience header: `asio.hpp`

5.137.5 ip::basic_resolver::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.137.5.1 ip::basic_resolver::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.137.5.2 ip::basic_resolver::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.137.6 ip::basic_resolver::get_io_service

Inherited from basic_io_object.

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the **io_service** object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the **io_service** object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.137.7 ip::basic_resolver::get_service

Get the service associated with the I/O object.

```
service_type & get_service();
```

```
const service_type & get_service() const;
```

5.137.7.1 ip::basic_resolver::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.137.7.2 ip::basic_resolver::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.137.8 ip::basic_resolver::implementation

Inherited from `basic_io_object`.

(Deprecated: Use `get_implementation()`) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.137.9 ip::basic_resolver::implementation_type

Inherited from `basic_io_object`.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: `asio/ip/basic_resolver.hpp`

Convenience header: `asio.hpp`

5.137.10 ip::basic_resolver::iterator

The iterator type.

```
typedef basic_resolver_iterator< InternetProtocol > iterator;
```

Types

Name	Description
<code>difference_type</code>	The type used for the distance between two iterators.
<code>iterator_category</code>	The iterator category.
<code>pointer</code>	The type of the result of applying operator->() to the iterator.
<code>reference</code>	The type of the result of applying operator*() to the iterator.
<code>value_type</code>	The type of the value pointed to by the iterator.

Member Functions

Name	Description
<code>basic_resolver_iterator</code>	Default constructor creates an end iterator.

Name	Description
create	Create an iterator from an addrinfo list returned by getaddrinfo. Create an iterator from an endpoint, host name and service name. Create an iterator from a sequence of endpoints, host and service name.
operator *	Dereference an iterator.
operator++	Increment operator (prefix). Increment operator (postfix).
operator->	Dereference an iterator.

Friends

Name	Description
operator!=	Test two iterators for inequality.
operator==	Test two iterators for equality.

The `ip::basic_resolver_iterator` class template is used to define iterators over the results returned by a resolver.

The iterator's value_type, obtained when the iterator is dereferenced, is:

```
const basic_resolver_entry<InternetProtocol>
```

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/basic_resolver.hpp`

Convenience header: `asio.hpp`

5.137.11 ip::basic_resolver::protocol_type

The protocol type.

```
typedef InternetProtocol protocol_type;
```

Requirements

Header: `asio/ip/basic_resolver.hpp`

Convenience header: `asio.hpp`

5.137.12 ip::basic_resolver::query

The query type.

```
typedef basic_resolver_query< InternetProtocol > query;
```

Types

Name	Description
flags	A bitmask type (C++ Std [lib.bitmask.types]).
protocol_type	The protocol type associated with the endpoint query.

Member Functions

Name	Description
basic_resolver_query	Construct with specified service name for any protocol. Construct with specified service name for a given protocol. Construct with specified host name and service name for any protocol. Construct with specified host name and service name for a given protocol.
hints	Get the hints associated with the query.
host_name	Get the host name associated with the query.
service_name	Get the service name associated with the query.

Data Members

Name	Description
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.
all_matching	If used with v4_mapped, return all matching IPv6 and IPv4 addresses.
canonical_name	Determine the canonical name of the host specified in the query.
numeric_host	Host name should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.
numeric_service	Service name should be treated as a numeric string defining a port number and no name resolution should be attempted.

Name	Description
passive	Indicate that returned endpoint is intended for use as a locally bound socket endpoint.
v4_mapped	If the query protocol family is specified as IPv6, return IPv4-mapped IPv6 addresses on finding no IPv6 addresses.

The `ip::basic_resolver_query` class template describes a query that can be passed to a resolver.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/basic_resolver.hpp`

Convenience header: `asio.hpp`

5.137.13 ip::basic_resolver::resolve

Perform forward resolution of a query to a list of entries.

```
iterator resolve(
    const query & q);

iterator resolve(
    const query & q,
    asio::error_code & ec);
```

Perform reverse resolution of an endpoint to a list of entries.

```
iterator resolve(
    const endpoint_type & e);

iterator resolve(
    const endpoint_type & e,
    asio::error_code & ec);
```

5.137.13.1 ip::basic_resolver::resolve (1 of 4 overloads)

Perform forward resolution of a query to a list of entries.

```
iterator resolve(
    const query & q);
```

This function is used to resolve a query into a list of endpoint entries.

Parameters

q A query object that determines what endpoints will be returned.

Return Value

A forward-only iterator that can be used to traverse the list of endpoint entries.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

A default constructed iterator represents the end of the list.

A successful call to this function is guaranteed to return at least one entry.

5.137.13.2 ip::basic_resolver::resolve (2 of 4 overloads)

Perform forward resolution of a query to a list of entries.

```
iterator resolve(
    const query & q,
    asio::error_code & ec);
```

This function is used to resolve a query into a list of endpoint entries.

Parameters

q A query object that determines what endpoints will be returned.

ec Set to indicate what error occurred, if any.

Return Value

A forward-only iterator that can be used to traverse the list of endpoint entries. Returns a default constructed iterator if an error occurs.

Remarks

A default constructed iterator represents the end of the list.

A successful call to this function is guaranteed to return at least one entry.

5.137.13.3 ip::basic_resolver::resolve (3 of 4 overloads)

Perform reverse resolution of an endpoint to a list of entries.

```
iterator resolve(
    const endpoint_type & e);
```

This function is used to resolve an endpoint into a list of endpoint entries.

Parameters

- e** An endpoint object that determines what endpoints will be returned.

Return Value

A forward-only iterator that can be used to traverse the list of endpoint entries.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

A default constructed iterator represents the end of the list.

A successful call to this function is guaranteed to return at least one entry.

5.137.13.4 ip::basic_resolver::resolve (4 of 4 overloads)

Perform reverse resolution of an endpoint to a list of entries.

```
iterator resolve(
    const endpoint_type & e,
    asio::error_code & ec);
```

This function is used to resolve an endpoint into a list of endpoint entries.

Parameters

- e** An endpoint object that determines what endpoints will be returned.

- ec** Set to indicate what error occurred, if any.

Return Value

A forward-only iterator that can be used to traverse the list of endpoint entries. Returns a default constructed iterator if an error occurs.

Remarks

A default constructed iterator represents the end of the list.

A successful call to this function is guaranteed to return at least one entry.

5.137.14 ip::basic_resolver::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.137.15 ip::basic_resolver::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef ResolverService service_type;
```

Requirements

Header: asio/ip/basic_resolver.hpp

Convenience header: asio.hpp

5.138 ip::basic_resolver_entry

An entry produced by a resolver.

```
template<
    typename InternetProtocol>
class basic_resolver_entry
```

Types

Name	Description
endpoint_type	The endpoint type associated with the endpoint entry.
protocol_type	The protocol type associated with the endpoint entry.

Member Functions

Name	Description
basic_resolver_entry	Default constructor. Construct with specified endpoint, host name and service name.
endpoint	Get the endpoint associated with the entry.
host_name	Get the host name associated with the entry.
operator endpoint_type	Convert to the endpoint associated with the entry.
service_name	Get the service name associated with the entry.

The `ip::basic_resolver_entry` class template describes an entry as returned by a resolver.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/basic_resolver_entry.hpp`

Convenience header: `asio.hpp`

5.138.1 `ip::basic_resolver_entry::basic_resolver_entry`

Default constructor.

```
basic_resolver_entry();
```

Construct with specified endpoint, host name and service name.

```
basic_resolver_entry(
    const endpoint_type & ep,
    const std::string & host,
    const std::string & service);
```

5.138.1.1 `ip::basic_resolver_entry::basic_resolver_entry (1 of 2 overloads)`

Default constructor.

```
basic_resolver_entry();
```

5.138.1.2 `ip::basic_resolver_entry::basic_resolver_entry (2 of 2 overloads)`

Construct with specified endpoint, host name and service name.

```
basic_resolver_entry(
    const endpoint_type & ep,
    const std::string & host,
    const std::string & service);
```

5.138.2 `ip::basic_resolver_entry::endpoint`

Get the endpoint associated with the entry.

```
endpoint_type endpoint() const;
```

5.138.3 `ip::basic_resolver_entry::endpoint_type`

The endpoint type associated with the endpoint entry.

```
typedef InternetProtocol::endpoint endpoint_type;
```

Requirements

Header: asio/ip/basic_resolver_entry.hpp

Convenience header: asio.hpp

5.138.4 ip::basic_resolver_entry::host_name

Get the host name associated with the entry.

```
std::string host_name() const;
```

5.138.5 ip::basic_resolver_entry::operator endpoint_type

Convert to the endpoint associated with the entry.

```
operator endpoint_type() const;
```

5.138.6 ip::basic_resolver_entry::protocol_type

The protocol type associated with the endpoint entry.

```
typedef InternetProtocol protocol_type;
```

Requirements

Header: asio/ip/basic_resolver_entry.hpp

Convenience header: asio.hpp

5.138.7 ip::basic_resolver_entry::service_name

Get the service name associated with the entry.

```
std::string service_name() const;
```

5.139 ip::basic_resolver_iterator

An iterator over the entries produced by a resolver.

```
template<
    typename InternetProtocol>
class basic_resolver_iterator
```

Name	Description
------	-------------

Types

Name	Description
difference_type	The type used for the distance between two iterators.
iterator_category	The iterator category.
pointer	The type of the result of applying operator->() to the iterator.
reference	The type of the result of applying operator*() to the iterator.
value_type	The type of the value pointed to by the iterator.

Member Functions

Name	Description
basic_resolver_iterator	Default constructor creates an end iterator.
create	Create an iterator from an addrinfo list returned by getaddrinfo. Create an iterator from an endpoint, host name and service name. Create an iterator from a sequence of endpoints, host and service name.
operator *	Dereference an iterator.
operator++	Increment operator (prefix). Increment operator (postfix).
operator->	Dereference an iterator.

Friends

Name	Description
operator!=	Test two iterators for inequality.
operator==	Test two iterators for equality.

The `ip::basic_resolver_iterator` class template is used to define iterators over the results returned by a resolver.

The iterator's `value_type`, obtained when the iterator is dereferenced, is:

```
const basic_resolver_entry<InternetProtocol>
```

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/basic_resolver_iterator.hpp

Convenience header: asio.hpp

5.139.1 ip::basic_resolver_iterator::basic_resolver_iterator

Default constructor creates an end iterator.

```
basic_resolver_iterator();
```

5.139.2 ip::basic_resolver_iterator::create

Create an iterator from an addrinfo list returned by getaddrinfo.

```
static basic_resolver_iterator create(
    asio::detail::addrinfo_type * address_info,
    const std::string & host_name,
    const std::string & service_name);
```

Create an iterator from an endpoint, host name and service name.

```
static basic_resolver_iterator create(
    const typename InternetProtocol::endpoint & endpoint,
    const std::string & host_name,
    const std::string & service_name);
```

Create an iterator from a sequence of endpoints, host and service name.

```
template<
    typename EndpointIterator>
static basic_resolver_iterator create(
    EndpointIterator begin,
    EndpointIterator end,
    const std::string & host_name,
    const std::string & service_name);
```

5.139.2.1 ip::basic_resolver_iterator::create (1 of 3 overloads)

Create an iterator from an addrinfo list returned by getaddrinfo.

```
static basic_resolver_iterator create(
    asio::detail::addrinfo_type * address_info,
    const std::string & host_name,
    const std::string & service_name);
```

5.139.2.2 ip::basic_resolver_iterator::create (2 of 3 overloads)

Create an iterator from an endpoint, host name and service name.

```
static basic_resolver_iterator create(
    const typename InternetProtocol::endpoint & endpoint,
    const std::string & host_name,
    const std::string & service_name);
```

5.139.2.3 ip::basic_resolver_iterator::create (3 of 3 overloads)

Create an iterator from a sequence of endpoints, host and service name.

```
template<
    typename EndpointIterator>
static basic_resolver_iterator create(
    EndpointIterator begin,
    EndpointIterator end,
    const std::string & host_name,
    const std::string & service_name);
```

5.139.3 ip::basic_resolver_iterator::difference_type

The type used for the distance between two iterators.

```
typedef std::ptrdiff_t difference_type;
```

Requirements

Header: asio/ip/basic_resolver_iterator.hpp

Convenience header: asio.hpp

5.139.4 ip::basic_resolver_iterator::iterator_category

The iterator category.

```
typedef std::forward_iterator_tag iterator_category;
```

Requirements

Header: asio/ip/basic_resolver_iterator.hpp

Convenience header: asio.hpp

5.139.5 ip::basic_resolver_iterator::operator *

Dereference an iterator.

```
const basic_resolver_entry< InternetProtocol > & operator *() const;
```

5.139.6 ip::basic_resolver_iterator::operator!=

Test two iterators for inequality.

```
friend bool operator!=(
    const basic_resolver_iterator & a,
    const basic_resolver_iterator & b);
```

Requirements

Header: asio/ip/basic_resolver_iterator.hpp

Convenience header: asio.hpp

5.139.7 ip::basic_resolver_iterator::operator++

Increment operator (prefix).

```
basic_resolver_iterator & operator++();
```

Increment operator (postfix).

```
basic_resolver_iterator operator++(
    int );
```

5.139.7.1 ip::basic_resolver_iterator::operator++ (1 of 2 overloads)

Increment operator (prefix).

```
basic_resolver_iterator & operator++();
```

5.139.7.2 ip::basic_resolver_iterator::operator++ (2 of 2 overloads)

Increment operator (postfix).

```
basic_resolver_iterator operator++(
    int );
```

5.139.8 ip::basic_resolver_iterator::operator->

Dereference an iterator.

```
const basic_resolver_entry< InternetProtocol > * operator->() const;
```

5.139.9 ip::basic_resolver_iterator::operator==

Test two iterators for equality.

```
friend bool operator==(

    const basic_resolver_iterator & a,
    const basic_resolver_iterator & b);
```

Requirements

Header: asio/ip/basic_resolver_iterator.hpp

Convenience header: asio.hpp

5.139.10 ip::basic_resolver_iterator::pointer

The type of the result of applying operator->() to the iterator.

```
typedef const basic_resolver_entry< InternetProtocol > * pointer;
```

Requirements

Header: asio/ip/basic_resolver_iterator.hpp

Convenience header: asio.hpp

5.139.11 ip::basic_resolver_iterator::reference

The type of the result of applying operator*() to the iterator.

```
typedef const basic_resolver_entry< InternetProtocol > & reference;
```

Types

Name	Description
endpoint_type	The endpoint type associated with the endpoint entry.
protocol_type	The protocol type associated with the endpoint entry.

Member Functions

Name	Description
basic_resolver_entry	Default constructor. Construct with specified endpoint, host name and service name.
endpoint	Get the endpoint associated with the entry.
host_name	Get the host name associated with the entry.
operator endpoint_type	Convert to the endpoint associated with the entry.
service_name	Get the service name associated with the entry.

The `ip::basic_resolver_entry` class template describes an entry as returned by a resolver.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/basic_resolver_iterator.hpp

Convenience header: asio.hpp

5.139.12 ip::basic_resolver_iterator::value_type

The type of the value pointed to by the iterator.

```
typedef basic_resolver_entry< InternetProtocol > value_type;
```

Types

Name	Description
endpoint_type	The endpoint type associated with the endpoint entry.
protocol_type	The protocol type associated with the endpoint entry.

Member Functions

Name	Description
basic_resolver_entry	Default constructor. Construct with specified endpoint, host name and service name.
endpoint	Get the endpoint associated with the entry.
host_name	Get the host name associated with the entry.
operator endpoint_type	Convert to the endpoint associated with the entry.
service_name	Get the service name associated with the entry.

The `ip::basic_resolver_entry` class template describes an entry as returned by a resolver.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/basic_resolver_iterator.hpp

Convenience header: asio.hpp

5.140 ip::basic_resolver_query

An query to be passed to a resolver.

```
template<
    typename InternetProtocol>
class basic_resolver_query :
    public ip::resolver_query_base
```

Types

Name	Description
flags	A bitmask type (C++ Std [lib.bitmask.types]).
protocol_type	The protocol type associated with the endpoint query.

Member Functions

Name	Description
basic_resolver_query	Construct with specified service name for any protocol. Construct with specified service name for a given protocol. Construct with specified host name and service name for any protocol. Construct with specified host name and service name for a given protocol.
hints	Get the hints associated with the query.
host_name	Get the host name associated with the query.
service_name	Get the service name associated with the query.

Data Members

Name	Description
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.
all_matching	If used with v4_mapped, return all matching IPv6 and IPv4 addresses.

Name	Description
canonical_name	Determine the canonical name of the host specified in the query.
numeric_host	Host name should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.
numeric_service	Service name should be treated as a numeric string defining a port number and no name resolution should be attempted.
passive	Indicate that returned endpoint is intended for use as a locally bound socket endpoint.
v4_mapped	If the query protocol family is specified as IPv6, return IPv4-mapped IPv6 addresses on finding no IPv6 addresses.

The `ip::basic_resolver_query` class template describes a query that can be passed to a resolver.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/basic_resolver_query.hpp`

Convenience header: `asio.hpp`

5.140.1 ip::basic_resolver_query::address_configured

Inherited from ip::resolver_query_base.

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.

```
static const flags address_configured = implementation_defined;
```

5.140.2 ip::basic_resolver_query::all_matching

Inherited from ip::resolver_query_base.

If used with v4_mapped, return all matching IPv6 and IPv4 addresses.

```
static const flags all_matching = implementation_defined;
```

5.140.3 ip::basic_resolver_query::basic_resolver_query

Construct with specified service name for any protocol.

```
basic_resolver_query(
    const std::string & service,
    resolver_query_base::flags resolve_flags = passive|address_configured);
```

Construct with specified service name for a given protocol.

```
basic_resolver_query(
    const protocol_type & protocol,
    const std::string & service,
    resolver_query_base::flags resolve_flags = passive|address_configured);
```

Construct with specified host name and service name for any protocol.

```
basic_resolver_query(
    const std::string & host,
    const std::string & service,
    resolver_query_base::flags resolve_flags = address_configured);
```

Construct with specified host name and service name for a given protocol.

```
basic_resolver_query(
    const protocol_type & protocol,
    const std::string & host,
    const std::string & service,
    resolver_query_base::flags resolve_flags = address_configured);
```

5.140.3.1 ip::basic_resolver_query::basic_resolver_query (1 of 4 overloads)

Construct with specified service name for any protocol.

```
basic_resolver_query(
    const std::string & service,
    resolver_query_base::flags resolve_flags = passive|address_configured);
```

This constructor is typically used to perform name resolution for local service binding.

Parameters

service A string identifying the requested service. This may be a descriptive name or a numeric string corresponding to a port number.

resolve_flags A set of flags that determine how name resolution should be performed. The default flags are suitable for local service binding.

Remarks

On POSIX systems, service names are typically defined in the file `/etc/services`. On Windows, service names may be found in the file `c:\windows\system32\drivers\etc\services`. Operating systems may use additional locations when resolving service names.

5.140.3.2 ip::basic_resolver_query::basic_resolver_query (2 of 4 overloads)

Construct with specified service name for a given protocol.

```
basic_resolver_query(
    const protocol_type & protocol,
    const std::string & service,
    resolver_query_base::flags resolve_flags = passive | address_configured);
```

This constructor is typically used to perform name resolution for local service binding with a specific protocol version.

Parameters

protocol A protocol object, normally representing either the IPv4 or IPv6 version of an internet protocol.

service A string identifying the requested service. This may be a descriptive name or a numeric string corresponding to a port number.

resolve_flags A set of flags that determine how name resolution should be performed. The default flags are suitable for local service binding.

Remarks

On POSIX systems, service names are typically defined in the file `/etc/services`. On Windows, service names may be found in the file `c:\windows\system32\drivers\etc\services`. Operating systems may use additional locations when resolving service names.

5.140.3.3 ip::basic_resolver_query::basic_resolver_query (3 of 4 overloads)

Construct with specified host name and service name for any protocol.

```
basic_resolver_query(
    const std::string & host,
    const std::string & service,
    resolver_query_base::flags resolve_flags = address_configured);
```

This constructor is typically used to perform name resolution for communication with remote hosts.

Parameters

host A string identifying a location. May be a descriptive name or a numeric address string. If an empty string and the passive flag has been specified, the resolved endpoints are suitable for local service binding. If an empty string and passive is not specified, the resolved endpoints will use the loopback address.

service A string identifying the requested service. This may be a descriptive name or a numeric string corresponding to a port number. May be an empty string, in which case all resolved endpoints will have a port number of 0.

resolve_flags A set of flags that determine how name resolution should be performed. The default flags are suitable for communication with remote hosts.

Remarks

On POSIX systems, host names may be locally defined in the file `/etc/hosts`. On Windows, host names may be defined in the file `c:\windows\system32\drivers\etc\hosts`. Remote host name resolution is performed using DNS. Operating systems may use additional locations when resolving host names (such as NETBIOS names on Windows).

On POSIX systems, service names are typically defined in the file `/etc/services`. On Windows, service names may be found in the file `c:\windows\system32\drivers\etc\services`. Operating systems may use additional locations when resolving service names.

5.140.3.4 ip::basic_resolver_query::basic_resolver_query (4 of 4 overloads)

Construct with specified host name and service name for a given protocol.

```
basic_resolver_query(
    const protocol_type & protocol,
    const std::string & host,
    const std::string & service,
    resolver_query_base::flags resolve_flags = address_configured);
```

This constructor is typically used to perform name resolution for communication with remote hosts.

Parameters

protocol A protocol object, normally representing either the IPv4 or IPv6 version of an internet protocol.

host A string identifying a location. May be a descriptive name or a numeric address string. If an empty string and the passive flag has been specified, the resolved endpoints are suitable for local service binding. If an empty string and passive is not specified, the resolved endpoints will use the loopback address.

service A string identifying the requested service. This may be a descriptive name or a numeric string corresponding to a port number. May be an empty string, in which case all resolved endpoints will have a port number of 0.

resolve_flags A set of flags that determine how name resolution should be performed. The default flags are suitable for communication with remote hosts.

Remarks

On POSIX systems, host names may be locally defined in the file /etc/hosts. On Windows, host names may be defined in the file c:\windows\system32\drivers\etc\hosts. Remote host name resolution is performed using DNS. Operating systems may use additional locations when resolving host names (such as NETBIOS names on Windows).

On POSIX systems, service names are typically defined in the file /etc/services. On Windows, service names may be found in the file c:\windows\system32\drivers\etc\services. Operating systems may use additional locations when resolving service names.

5.140.4 ip::basic_resolver_query::canonical_name

Inherited from ip::resolver_query_base.

Determine the canonical name of the host specified in the query.

```
static const flags canonical_name = implementation_defined;
```

5.140.5 ip::basic_resolver_query::flags

Inherited from ip::resolver_query_base.

A bitmask type (C++ Std [lib.bitmask.types]).

```
typedef unspecified flags;
```

Requirements

Header: asio/ip/basic_resolver_query.hpp

Convenience header: asio.hpp

5.140.6 ip::basic_resolver_query::hints

Get the hints associated with the query.

```
const asio::detail::addrinfo_type & hints() const;
```

5.140.7 ip::basic_resolver_query::host_name

Get the host name associated with the query.

```
std::string host_name() const;
```

5.140.8 ip::basic_resolver_query::numeric_host

Inherited from ip::resolver_query_base.

Host name should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.

```
static const flags numeric_host = implementation_defined;
```

5.140.9 ip::basic_resolver_query::numeric_service

Inherited from ip::resolver_query_base.

Service name should be treated as a numeric string defining a port number and no name resolution should be attempted.

```
static const flags numeric_service = implementation_defined;
```

5.140.10 ip::basic_resolver_query::passive

Inherited from ip::resolver_query_base.

Indicate that returned endpoint is intended for use as a locally bound socket endpoint.

```
static const flags passive = implementation_defined;
```

5.140.11 ip::basic_resolver_query::protocol_type

The protocol type associated with the endpoint query.

```
typedef InternetProtocol protocol_type;
```

Requirements

Header: asio/ip/basic_resolver_query.hpp

Convenience header: asio.hpp

5.140.12 ip::basic_resolver_query::service_name

Get the service name associated with the query.

```
std::string service_name() const;
```

5.140.13 ip::basic_resolver_query::v4_mapped

Inherited from `ip::resolver_query_base`.

If the query protocol family is specified as IPv6, return IPv4-mapped IPv6 addresses on finding no IPv6 addresses.

```
static const flags v4_mapped = implementation_defined;
```

5.141 ip::host_name

Get the current host name.

```
std::string host_name();  
  
std::string host_name(  
    asio::error_code & ec);
```

Requirements

Header: `asio/ip/host_name.hpp`

Convenience header: `asio.hpp`

5.141.1 ip::host_name (1 of 2 overloads)

Get the current host name.

```
std::string host_name();
```

5.141.2 ip::host_name (2 of 2 overloads)

Get the current host name.

```
std::string host_name(  
    asio::error_code & ec);
```

5.142 ip::icmp

Encapsulates the flags needed for ICMP.

```
class icmp
```

Types

Name	Description
endpoint	The type of a ICMP endpoint.
resolver	The ICMP resolver type.
socket	The ICMP socket type.

Member Functions

Name	Description
family	Obtain an identifier for the protocol family.
protocol	Obtain an identifier for the protocol.
type	Obtain an identifier for the type of the protocol.
v4	Construct to represent the IPv4 ICMP protocol.
v6	Construct to represent the IPv6 ICMP protocol.

Friends

Name	Description
operator!=	Compare two protocols for inequality.
operator==	Compare two protocols for equality.

The `ip::icmp` class contains flags necessary for ICMP sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: `asio/ip/icmp.hpp`

Convenience header: `asio.hpp`

5.142.1 ip::icmp::endpoint

The type of a ICMP endpoint.

```
typedef basic_endpoint< icmp > endpoint;
```

Types

Name	Description
data_type	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
protocol_type	The protocol type associated with the endpoint.

Member Functions

Name	Description
address	Get the IP address associated with the endpoint. Set the IP address associated with the endpoint.
basic_endpoint	Default constructor. Construct an endpoint using a port number, specified in the host's byte order. The IP address will be the any address (i.e. INADDR_ANY or in6addr_any). This constructor would typically be used for accepting new connections. Construct an endpoint using a port number and an IP address. This constructor may be used for accepting connections on a specific interface or for making a connection to a remote endpoint. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
port	Get the port associated with the endpoint. The port number is always in the host's byte order. Set the port associated with the endpoint. The port number is always in the host's byte order.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

Related Functions

Name	Description
operator<<	Output an endpoint as a string.

The `ip::basic_endpoint` class template describes an endpoint that may be associated with a particular socket.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/icmp.hpp`

Convenience header: `asio.hpp`

5.142.2 ip::icmp::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.142.3 ip::icmp::operator!=

Compare two protocols for inequality.

```
friend bool operator!=(
    const icmp & p1,
    const icmp & p2);
```

Requirements

Header: `asio/ip/icmp.hpp`

Convenience header: `asio.hpp`

5.142.4 ip::icmp::operator==

Compare two protocols for equality.

```
friend bool operator==(

    const icmp & p1,
    const icmp & p2);
```

Requirements

Header: `asio/ip/icmp.hpp`

Convenience header: `asio.hpp`

5.142.5 ip::icmp::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.142.6 ip::icmp::resolver

The ICMP resolver type.

```
typedef basic_resolver< icmp > resolver;
```

Types

Name	Description
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
iterator	The iterator type.
protocol_type	The protocol type.
query	The query type.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
async_resolve	Asynchronously perform forward resolution of a query to a list of entries. Asynchronously perform reverse resolution of an endpoint to a list of entries.
basic_resolver	Constructor.
cancel	Cancel any asynchronous operations that are waiting on the resolver.
get_io_service	Get the io_service associated with the object.
resolve	Perform forward resolution of a query to a list of entries. Perform reverse resolution of an endpoint to a list of entries.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `ip::basic_resolver` class template provides the ability to resolve a query to a list of endpoints.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/icmp.hpp`

Convenience header: `asio.hpp`

5.142.7 ip::icmp::socket

The ICMP socket type.

```
typedef basic_raw_socket< icmp > socket;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.

Name	Description
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive on a connected socket.
async_receive_from	Start an asynchronous receive.

Name	Description
async_send	Start an asynchronous send on a connected socket.
async_send_to	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_raw_socket	Construct a basic_raw_socket without opening it. Construct and open a basic_raw_socket. Construct a basic_raw_socket, opening it and binding it to the given local endpoint. Construct a basic_raw_socket on an existing native socket. Move-construct a basic_raw_socket from another. Move-construct a basic_raw_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.

Name	Description
operator=	Move-assign a basic_raw_socket from another. Move-assign a basic_raw_socket from a socket of another protocol type.
receive	Receive some data on a connected socket.
receive_from	Receive raw data with the endpoint of the sender.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on a connected socket.
send_to	Send raw data to the specified endpoint.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_raw_socket` class template provides asynchronous and blocking raw-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/icmp.hpp`

Convenience header: `asio.hpp`

5.142.8 `ip::icmp::type`

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.142.9 `ip::icmp::v4`

Construct to represent the IPv4 ICMP protocol.

```
static icmp v4();
```

5.142.10 `ip::icmp::v6`

Construct to represent the IPv6 ICMP protocol.

```
static icmp v6();
```

5.143 `ip::multicast::enable_loopback`

Socket option determining whether outgoing multicast packets will be received on the same socket if it is a member of the multicast group.

```
typedef implementation_defined enable_loopback;
```

Implements the `IPPROTO_IP/IP_MULTICAST_LOOP` socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::ip::multicast::enable_loopback option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::ip::multicast::enable_loopback option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/ip/multicast.hpp

Convenience header: asio.hpp

5.144 ip::multicast::hops

Socket option for time-to-live associated with outgoing multicast packets.

```
typedef implementation_defined hops;
```

Implements the IPPROTO_IP/IP_MULTICAST_TTL socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::ip::multicast::hops option(4);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::ip::multicast::hops option;
socket.get_option(option);
int ttl = option.value();
```

Requirements

Header: asio/ip/multicast.hpp

Convenience header: asio.hpp

5.145 ip::multicast::join_group

Socket option to join a multicast group on a specified interface.

```
typedef implementation_defined join_group;
```

Implements the IPPROTO_IP/IP_ADD_MEMBERSHIP socket option.

Examples

Setting the option to join a multicast group:

```
asio::ip::udp::socket socket(io_service);
...
asio::ip::address multicast_address =
    asio::ip::address::from_string("225.0.0.1");
asio::ip::multicast::join_group option(multicast_address);
socket.set_option(option);
```

Requirements

Header: asio/ip/multicast.hpp

Convenience header: asio.hpp

5.146 ip::multicast::leave_group

Socket option to leave a multicast group on a specified interface.

```
typedef implementation_defined leave_group;
```

Implements the IPPROTO_IP/IP_DROP_MEMBERSHIP socket option.

Examples

Setting the option to leave a multicast group:

```
asio::ip::udp::socket socket(io_service);
...
asio::ip::address multicast_address =
    asio::ip::address::from_string("225.0.0.1");
asio::ip::multicast::leave_group option(multicast_address);
socket.set_option(option);
```

Requirements

Header: asio/ip/multicast.hpp

Convenience header: asio.hpp

5.147 ip::multicast::outbound_interface

Socket option for local interface to use for outgoing multicast packets.

```
typedef implementation_defined outbound_interface;
```

Implements the IPPROTO_IP/IP_MULTICAST_IF socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::ip::address_v4 local_interface =
    asio::ip::address_v4::from_string("1.2.3.4");
asio::ip::multicast::outbound_interface option(local_interface);
socket.set_option(option);
```

Requirements

Header: asio/ip/multicast.hpp

Convenience header: asio.hpp

5.148 ip::resolver_query_base

The `ip::resolver_query_base` class is used as a base for the `ip::basic_resolver_query` class templates to provide a common place to define the flag constants.

```
class resolver_query_base
```

Types

Name	Description
<code>flags</code>	A bitmask type (C++ Std [lib.bitmask.types]).

Protected Member Functions

Name	Description
<code>~resolver_query_base</code>	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
<code>address_configured</code>	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.
<code>all_matching</code>	If used with <code>v4_mapped</code> , return all matching IPv6 and IPv4 addresses.
<code>canonical_name</code>	Determine the canonical name of the host specified in the query.

Name	Description
numeric_host	Host name should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.
numeric_service	Service name should be treated as a numeric string defining a port number and no name resolution should be attempted.
passive	Indicate that returned endpoint is intended for use as a locally bound socket endpoint.
v4_mapped	If the query protocol family is specified as IPv6, return IPv4-mapped IPv6 addresses on finding no IPv6 addresses.

Requirements

Header: asio/ip/resolver_query_base.hpp

Convenience header: asio.hpp

5.148.1 ip::resolver_query_base::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.

```
static const flags address_configured = implementation_defined;
```

5.148.2 ip::resolver_query_base::all_matching

If used with v4_mapped, return all matching IPv6 and IPv4 addresses.

```
static const flags all_matching = implementation_defined;
```

5.148.3 ip::resolver_query_base::canonical_name

Determine the canonical name of the host specified in the query.

```
static const flags canonical_name = implementation_defined;
```

5.148.4 ip::resolver_query_base::flags

A bitmask type (C++ Std [lib.bitmask.types]).

```
typedef unspecified flags;
```

Requirements

Header: asio/ip/resolver_query_base.hpp

Convenience header: asio.hpp

5.148.5 ip::resolver_query_base::numeric_host

Host name should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.

```
static const flags numeric_host = implementation_defined;
```

5.148.6 ip::resolver_query_base::numeric_service

Service name should be treated as a numeric string defining a port number and no name resolution should be attempted.

```
static const flags numeric_service = implementation_defined;
```

5.148.7 ip::resolver_query_base::passive

Indicate that returned endpoint is intended for use as a locally bound socket endpoint.

```
static const flags passive = implementation_defined;
```

5.148.8 ip::resolver_query_base::v4_mapped

If the query protocol family is specified as IPv6, return IPv4-mapped IPv6 addresses on finding no IPv6 addresses.

```
static const flags v4_mapped = implementation_defined;
```

5.148.9 ip::resolver_query_base::~resolver_query_base

Protected destructor to prevent deletion through this type.

```
~resolver_query_base();
```

5.149 ip::resolver_service

Default service implementation for a resolver.

```
template<
    typename InternetProtocol>
class resolver_service :
    public io_service::service
```

Types

Name	Description
endpoint_type	The endpoint type.
implementation_type	The type of a resolver implementation.
iterator_type	The iterator type.
protocol_type	The protocol type.
query_type	The query type.

Member Functions

Name	Description
async_resolve	Asynchronously resolve a query to a list of entries. Asynchronously resolve an endpoint to a list of entries.
cancel	Cancel pending asynchronous operations.
construct	Construct a new resolver implementation.
destroy	Destroy a resolver implementation.
get_io_service	Get the io_service object that owns the service.
resolve	Resolve a query to a list of entries. Resolve an endpoint to a list of entries.
resolver_service	Construct a new resolver service for the specified io_service.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/ip/resolver_service.hpp

Convenience header: asio.hpp

5.149.1 ip::resolver_service::async_resolve

Asynchronously resolve a query to a list of entries.

```
template<
    typename ResolveHandler>
void-or-deduced async_resolve(
    implementation_type & impl,
    const query_type & query,
    ResolveHandler handler);
```

Asynchronously resolve an endpoint to a list of entries.

```
template<
    typename ResolveHandler>
void-or-deduced async_resolve(
    implementation_type & impl,
    const endpoint_type & endpoint,
    ResolveHandler handler);
```

5.149.1.1 ip::resolver_service::async_resolve (1 of 2 overloads)

Asynchronously resolve a query to a list of entries.

```
template<
    typename ResolveHandler>
void-or-deduced async_resolve(
    implementation_type & impl,
    const query_type & query,
    ResolveHandler handler);
```

5.149.1.2 ip::resolver_service::async_resolve (2 of 2 overloads)

Asynchronously resolve an endpoint to a list of entries.

```
template<
    typename ResolveHandler>
void-or-deduced async_resolve(
    implementation_type & impl,
    const endpoint_type & endpoint,
    ResolveHandler handler);
```

5.149.2 ip::resolver_service::cancel

Cancel pending asynchronous operations.

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

5.149.3 ip::resolver_service::construct

Construct a new resolver implementation.

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

5.149.4 ip::resolver_service::destroy

Destroy a resolver implementation.

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

5.149.5 ip::resolver_service::endpoint_type

The endpoint type.

```
typedef InternetProtocol::endpoint endpoint_type;
```

Requirements

Header: asio/ip/resolver_service.hpp

Convenience header: asio.hpp

5.149.6 ip::resolver_service::get_io_service

Inherited from io_service.

Get the `io_service` object that owns the service.

```
asio::io_service & get_io_service();
```

5.149.7 ip::resolver_service::id

The unique service identifier.

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

5.149.8 ip::resolver_service::implementation_type

The type of a resolver implementation.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: `asio/ip/resolver_service.hpp`

Convenience header: `asio.hpp`

5.149.9 ip::resolver_service::iterator_type

The iterator type.

```
typedef basic_resolver_iterator< InternetProtocol > iterator_type;
```

Types

Name	Description
<code>difference_type</code>	The type used for the distance between two iterators.
<code>iterator_category</code>	The iterator category.
<code>pointer</code>	The type of the result of applying operator->() to the iterator.
<code>reference</code>	The type of the result of applying operator*() to the iterator.
<code>value_type</code>	The type of the value pointed to by the iterator.

Member Functions

Name	Description
basic_resolver_iterator	Default constructor creates an end iterator.
create	Create an iterator from an addrinfo list returned by getaddrinfo. Create an iterator from an endpoint, host name and service name. Create an iterator from a sequence of endpoints, host and service name.
operator *	Dereference an iterator.
operator++	Increment operator (prefix). Increment operator (postfix).
operator->	Dereference an iterator.

Friends

Name	Description
operator!=	Test two iterators for inequality.
operator==	Test two iterators for equality.

The `ip::basic_resolver_iterator` class template is used to define iterators over the results returned by a resolver.

The iterator's value_type, obtained when the iterator is dereferenced, is:

```
const basic_resolver_entry<InternetProtocol>
```

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/resolver_service.hpp`

Convenience header: `asio.hpp`

5.149.10 ip::resolver_service::protocol_type

The protocol type.

```
typedef InternetProtocol protocol_type;
```

Requirements

Header: asio/ip/resolver_service.hpp

Convenience header: asio.hpp

5.149.11 ip::resolver_service::query_type

The query type.

```
typedef basic_resolver_query< InternetProtocol > query_type;
```

Types

Name	Description
flags	A bitmask type (C++ Std [lib.bitmask.types]).
protocol_type	The protocol type associated with the endpoint query.

Member Functions

Name	Description
basic_resolver_query	Construct with specified service name for any protocol. Construct with specified service name for a given protocol. Construct with specified host name and service name for any protocol. Construct with specified host name and service name for a given protocol.
hints	Get the hints associated with the query.
host_name	Get the host name associated with the query.
service_name	Get the service name associated with the query.

Data Members

Name	Description
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.
all_matching	If used with v4_mapped, return all matching IPv6 and IPv4 addresses.
canonical_name	Determine the canonical name of the host specified in the query.

Name	Description
numeric_host	Host name should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.
numeric_service	Service name should be treated as a numeric string defining a port number and no name resolution should be attempted.
passive	Indicate that returned endpoint is intended for use as a locally bound socket endpoint.
v4_mapped	If the query protocol family is specified as IPv6, return IPv4-mapped IPv6 addresses on finding no IPv6 addresses.

The `ip::basic_resolver_query` class template describes a query that can be passed to a resolver.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/resolver_service.hpp`

Convenience header: `asio.hpp`

5.149.12 ip::resolver_service::resolve

Resolve a query to a list of entries.

```
iterator_type resolve(
    implementation_type & impl,
    const query_type & query,
    asio::error_code & ec);
```

Resolve an endpoint to a list of entries.

```
iterator_type resolve(
    implementation_type & impl,
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.149.12.1 ip::resolver_service::resolve (1 of 2 overloads)

Resolve a query to a list of entries.

```
iterator_type resolve(
    implementation_type & impl,
    const query_type & query,
    asio::error_code & ec);
```

5.149.12.2 ip::resolver_service::resolve (2 of 2 overloads)

Resolve an endpoint to a list of entries.

```
iterator_type resolve(
    implementation_type & impl,
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.149.13 ip::resolver_service::resolver_service

Construct a new resolver service for the specified [io_service](#).

```
resolver_service(
    asio::io_service & io_service);
```

5.150 ip::tcp

Encapsulates the flags needed for TCP.

```
class tcp
```

Types

Name	Description
acceptor	The TCP acceptor type.
endpoint	The type of a TCP endpoint.
iostream	The TCP iostream type.
no_delay	Socket option for disabling the Nagle algorithm.
resolver	The TCP resolver type.
socket	The TCP socket type.

Member Functions

Name	Description
family	Obtain an identifier for the protocol family.
protocol	Obtain an identifier for the protocol.
type	Obtain an identifier for the type of the protocol.
v4	Construct to represent the IPv4 TCP protocol.
v6	Construct to represent the IPv6 TCP protocol.

Friends

Name	Description
operator!=	Compare two protocols for inequality.
operator==	Compare two protocols for equality.

The `ip::tcp` class contains flags necessary for TCP sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: `asio/ip/tcp.hpp`

Convenience header: `asio.hpp`

5.150.1 ip::tcp::acceptor

The TCP acceptor type.

```
typedef basic_socket_acceptor< tcp > acceptor;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.

Name	Description
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of an acceptor.
native_type	(Deprecated: Use native_handle_type.) The native representation of an acceptor.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
accept	Accept a new connection. Accept a new connection and obtain the endpoint of the peer.
assign	Assigns an existing native acceptor to the acceptor.
async_accept	Start an asynchronous accept.
basic_socket_acceptor	Construct an acceptor without opening it. Construct an open acceptor. Construct an acceptor opened on the given endpoint. Construct a basic_socket_acceptor on an existing native acceptor. Move-construct a basic_socket_acceptor from another. Move-construct a basic_socket_acceptor from an acceptor of another protocol type.
bind	Bind the acceptor to the given local endpoint.

Name	Description
cancel	Cancel all asynchronous operations associated with the acceptor.
close	Close the acceptor.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the acceptor.
io_control	Perform an IO control command on the acceptor.
is_open	Determine whether the acceptor is open.
listen	Place the acceptor into the state where it will listen for new connections.
local_endpoint	Get the local endpoint of the acceptor.
native	(Deprecated: Use native_handle().) Get the native acceptor representation.
native_handle	Get the native acceptor representation.
native_non_blocking	Gets the non-blocking mode of the native acceptor implementation. Sets the non-blocking mode of the native acceptor implementation.
non_blocking	Gets the non-blocking mode of the acceptor. Sets the non-blocking mode of the acceptor.
open	Open the acceptor using the specified protocol.
operator=	Move-assign a basic_socket_acceptor from another. Move-assign a basic_socket_acceptor from an acceptor of another protocol type.
set_option	Set an option on the acceptor.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_socket_acceptor` class template is used for accepting new socket connections.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Example

Opening a socket acceptor with the SO_REUSEADDR option enabled:

```
asio::ip::tcp::acceptor acceptor(io_service);
asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), port);
acceptor.open(endpoint.protocol());
acceptor.set_option(asio::ip::tcp::acceptor::reuse_address(true));
acceptor.bind(endpoint);
acceptor.listen();
```

Requirements

Header: `asio/ip/tcp.hpp`

Convenience header: `asio.hpp`

5.150.2 ip::tcp::endpoint

The type of a TCP endpoint.

```
typedef basic_endpoint< tcp > endpoint;
```

Types

Name	Description
data_type	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
protocol_type	The protocol type associated with the endpoint.

Member Functions

Name	Description
address	Get the IP address associated with the endpoint. Set the IP address associated with the endpoint.
basic_endpoint	Default constructor. Construct an endpoint using a port number, specified in the host's byte order. The IP address will be the any address (i.e. INADDR_ANY or in6addr_any). This constructor would typically be used for accepting new connections. Construct an endpoint using a port number and an IP address. This constructor may be used for accepting connections on a specific interface or for making a connection to a remote endpoint. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
port	Get the port associated with the endpoint. The port number is always in the host's byte order. Set the port associated with the endpoint. The port number is always in the host's byte order.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.

Name	Description
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

Related Functions

Name	Description
operator<<	Output an endpoint as a string.

The `ip::basic_endpoint` class template describes an endpoint that may be associated with a particular socket.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/tcp.hpp`

Convenience header: `asio.hpp`

5.150.3 ip::tcp::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.150.4 ip::tcp::iostream

The TCP iostream type.

```
typedef basic_socket_iostream< tcp > iostream;
```

Types

Name	Description
duration_type	The duration type.
endpoint_type	The endpoint type.
time_type	The time type.

Member Functions

Name	Description
basic_socket_iostream	Construct a basic_socket_iostream without establishing a connection. Establish a connection to an endpoint corresponding to a resolver query.
close	Close the connection.
connect	Establish a connection to an endpoint corresponding to a resolver query.
error	Get the last error associated with the stream.
expires_at	Get the stream's expiry time as an absolute time. Set the stream's expiry time as an absolute time.
expires_from_now	Get the timer's expiry time relative to now. Set the stream's expiry time relative to now.
rdbuf	Return a pointer to the underlying streambuf.

Requirements

Header: asio/ip/tcp.hpp

Convenience header: asio.hpp

5.150.5 ip::tcp::no_delay

Socket option for disabling the Nagle algorithm.

```
typedef implementation_defined no_delay;
```

Implements the IPPROTO_TCP/TCP_NODELAY socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::tcp::no_delay option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/ip/tcp.hpp

Convenience header: asio.hpp

5.150.6 ip::tcp::operator!=

Compare two protocols for inequality.

```
friend bool operator!=(
    const tcp & p1,
    const tcp & p2);
```

Requirements

Header: asio/ip/tcp.hpp

Convenience header: asio.hpp

5.150.7 ip::tcp::operator==

Compare two protocols for equality.

```
friend bool operator==((
    const tcp & p1,
    const tcp & p2);
```

Requirements

Header: asio/ip/tcp.hpp

Convenience header: asio.hpp

5.150.8 ip::tcp::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.150.9 ip::tcp::resolver

The TCP resolver type.

```
typedef basic_resolver< tcp > resolver;
```

Name	Description
------	-------------

Types

Name	Description
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
iterator	The iterator type.
protocol_type	The protocol type.
query	The query type.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
async_resolve	Asynchronously perform forward resolution of a query to a list of entries. Asynchronously perform reverse resolution of an endpoint to a list of entries.
basic_resolver	Constructor.
cancel	Cancel any asynchronous operations that are waiting on the resolver.
get_io_service	Get the io_service associated with the object.
resolve	Perform forward resolution of a query to a list of entries. Perform reverse resolution of an endpoint to a list of entries.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `ip::basic_resolver` class template provides the ability to resolve a query to a list of endpoints.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/tcp.hpp`

Convenience header: `asio.hpp`

5.150.10 ip::tcp::socket

The TCP socket type.

```
typedef basic_stream_socket< tcp > socket;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.

Name	Description
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_read_some	Start an asynchronous read.
async_receive	Start an asynchronous receive.
async_send	Start an asynchronous send.
async_write_some	Start an asynchronous write.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.

Name	Description
<code>basic_stream_socket</code>	Construct a <code>basic_stream_socket</code> without opening it. Construct and open a <code>basic_stream_socket</code> . Construct a <code>basic_stream_socket</code> , opening it and binding it to the given local endpoint. Construct a <code>basic_stream_socket</code> on an existing native socket. Move-construct a <code>basic_stream_socket</code> from another. Move-construct a <code>basic_stream_socket</code> from a socket of another protocol type.
<code>bind</code>	Bind the socket to the given local endpoint.
<code>cancel</code>	Cancel all asynchronous operations associated with the socket.
<code>close</code>	Close the socket.
<code>connect</code>	Connect the socket to the specified endpoint.
<code>get_io_service</code>	Get the <code>io_service</code> associated with the object.
<code>get_option</code>	Get an option from the socket.
<code>io_control</code>	Perform an IO control command on the socket.
<code>is_open</code>	Determine whether the socket is open.
<code>local_endpoint</code>	Get the local endpoint of the socket.
<code>lowest_layer</code>	Get a reference to the lowest layer. Get a const reference to the lowest layer.
<code>native</code>	(Deprecated: Use <code>native_handle()</code>) Get the native socket representation.
<code>native_handle</code>	Get the native socket representation.
<code>native_non_blocking</code>	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
<code>non_blocking</code>	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
<code>open</code>	Open the socket using the specified protocol.
<code>operator=</code>	Move-assign a <code>basic_stream_socket</code> from another. Move-assign a <code>basic_stream_socket</code> from a socket of another protocol type.
<code>read_some</code>	Read some data from the socket.
<code>receive</code>	Receive some data on the socket. Receive some data on a connected socket.

Name	Description
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on the socket.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.
write_some	Write some data to the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_stream_socket` class template provides asynchronous and blocking stream-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/tcp.hpp

Convenience header: asio.hpp

5.150.11 ip::tcp::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.150.12 ip::tcp::v4

Construct to represent the IPv4 TCP protocol.

```
static tcp v4();
```

5.150.13 ip::tcp::v6

Construct to represent the IPv6 TCP protocol.

```
static tcp v6();
```

5.151 ip::udp

Encapsulates the flags needed for UDP.

```
class udp
```

Types

Name	Description
endpoint	The type of a UDP endpoint.
resolver	The UDP resolver type.
socket	The UDP socket type.

Member Functions

Name	Description
family	Obtain an identifier for the protocol family.
protocol	Obtain an identifier for the protocol.
type	Obtain an identifier for the type of the protocol.
v4	Construct to represent the IPv4 UDP protocol.
v6	Construct to represent the IPv6 UDP protocol.

Friends

Name	Description
operator!=	Compare two protocols for inequality.
operator==	Compare two protocols for equality.

The `ip::udp` class contains flags necessary for UDP sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: `asio/ip/udp.hpp`

Convenience header: `asio.hpp`

5.151.1 ip::udp::endpoint

The type of a UDP endpoint.

```
typedef basic_endpoint< udp > endpoint;
```

Types

Name	Description
data_type	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
protocol_type	The protocol type associated with the endpoint.

Member Functions

Name	Description
address	Get the IP address associated with the endpoint. Set the IP address associated with the endpoint.
basic_endpoint	Default constructor. Construct an endpoint using a port number, specified in the host's byte order. The IP address will be the any address (i.e. INADDR_ANY or in6addr_any). This constructor would typically be used for accepting new connections. Construct an endpoint using a port number and an IP address. This constructor may be used for accepting connections on a specific interface or for making a connection to a remote endpoint. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
port	Get the port associated with the endpoint. The port number is always in the host's byte order. Set the port associated with the endpoint. The port number is always in the host's byte order.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

Related Functions

Name	Description
operator<<	Output an endpoint as a string.

The `ip::basic_endpoint` class template describes an endpoint that may be associated with a particular socket.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/udp.hpp`

Convenience header: `asio.hpp`

5.151.2 ip::udp::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.151.3 ip::udp::operator!=

Compare two protocols for inequality.

```
friend bool operator!=(
    const udp & p1,
    const udp & p2);
```

Requirements

Header: `asio/ip/udp.hpp`

Convenience header: `asio.hpp`

5.151.4 ip::udp::operator==

Compare two protocols for equality.

```
friend bool operator==(

    const udp & p1,
    const udp & p2);
```

Requirements

Header: `asio/ip/udp.hpp`

Convenience header: `asio.hpp`

5.151.5 ip::udp::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.151.6 ip::udp::resolver

The UDP resolver type.

```
typedef basic_resolver< udp > resolver;
```

Types

Name	Description
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
iterator	The iterator type.
protocol_type	The protocol type.
query	The query type.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
async_resolve	Asynchronously perform forward resolution of a query to a list of entries. Asynchronously perform reverse resolution of an endpoint to a list of entries.
basic_resolver	Constructor.
cancel	Cancel any asynchronous operations that are waiting on the resolver.
get_io_service	Get the io_service associated with the object.
resolve	Perform forward resolution of a query to a list of entries. Perform reverse resolution of an endpoint to a list of entries.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `ip::basic_resolver` class template provides the ability to resolve a query to a list of endpoints.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/udp.hpp`

Convenience header: `asio.hpp`

5.151.7 ip::udp::socket

The UDP socket type.

```
typedef basic_datagram_socket< udp > socket;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.

Name	Description
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive on a connected socket.
async_receive_from	Start an asynchronous receive.

Name	Description
async_send	Start an asynchronous send on a connected socket.
async_send_to	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_datagram_socket	Construct a basic_datagram_socket without opening it. Construct and open a basic_datagram_socket. Construct a basic_datagram_socket, opening it and binding it to the given local endpoint. Construct a basic_datagram_socket on an existing native socket. Move-construct a basic_datagram_socket from another. Move-construct a basic_datagram_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.

Name	Description
operator=	Move-assign a basic_datagram_socket from another. Move-assign a basic_datagram_socket from a socket of another protocol type.
receive	Receive some data on a connected socket.
receive_from	Receive a datagram with the endpoint of the sender.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on a connected socket.
send_to	Send a datagram to the specified endpoint.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_datagram_socket` class template provides asynchronous and blocking datagram-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/udp.hpp`

Convenience header: `asio.hpp`

5.151.8 ip::udp::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.151.9 ip::udp::v4

Construct to represent the IPv4 UDP protocol.

```
static udp v4();
```

5.151.10 ip::udp::v6

Construct to represent the IPv6 UDP protocol.

```
static udp v6();
```

5.152 ip::unicast::hops

Socket option for time-to-live associated with outgoing unicast packets.

```
typedef implementation_defined hops;
```

Implements the IPPROTO_IP/IP_UNICAST_TTL socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::ip::unicast::hops option(4);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::ip::unicast::hops option;
socket.get_option(option);
int ttl = option.value();
```

Requirements

Header: asio/ip/unicast.hpp

Convenience header: asio.hpp

5.153 ip::v6_only

Socket option for determining whether an IPv6 socket supports IPv6 communication only.

```
typedef implementation_defined v6_only;
```

Implements the IPPROTO_IPV6/IP_V6ONLY socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::v6_only option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::ip::v6_only option;
socket.get_option(option);
bool v6_only = option.value();
```

Requirements

Header: asio/ip/v6_only.hpp

Convenience header: asio.hpp

5.154 is_match_condition

Type trait used to determine whether a type can be used as a match condition function with `read_until` and `async_read_until`.

```
template<
    typename T>
struct is_match_condition
```

Data Members

Name	Description
value	The value member is true if the type may be used as a match condition.

Requirements

Header: `asio/read_until.hpp`

Convenience header: `asio.hpp`

5.154.1 is_match_condition::value

The value member is true if the type may be used as a match condition.

```
static const bool value;
```

5.155 is_read_buffered

The `is_read_buffered` class is a traits class that may be used to determine whether a stream type supports buffering of read data.

```
template<
    typename Stream>
class is_read_buffered
```

Data Members

Name	Description
value	The value member is true only if the Stream type supports buffering of read data.

Requirements

Header: `asio/is_read_buffered.hpp`

Convenience header: `asio.hpp`

5.155.1 `is_read_buffered::value`

The value member is true only if the Stream type supports buffering of read data.

```
static const bool value;
```

5.156 `is_write_buffered`

The `is_write_buffered` class is a traits class that may be used to determine whether a stream type supports buffering of written data.

```
template<
    typename Stream>
class is_write_buffered
```

Data Members

Name	Description
<code>value</code>	The value member is true only if the Stream type supports buffering of written data.

Requirements

Header: `asio/is_write_buffered.hpp`

Convenience header: `asio.hpp`

5.156.1 `is_write_buffered::value`

The value member is true only if the Stream type supports buffering of written data.

```
static const bool value;
```

5.157 `local::basic_endpoint`

Describes an endpoint for a UNIX socket.

```
template<
    typename Protocol>
class basic_endpoint
```

Types

Name	Description
<code>data_type</code>	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
<code>protocol_type</code>	The protocol type associated with the endpoint.

Member Functions

Name	Description
basic_endpoint	Default constructor. Construct an endpoint using the specified path name. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
path	Get the path associated with the endpoint. Set the path associated with the endpoint.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

Related Functions

Name	Description
operator<<	Output an endpoint as a string.

The `local::basic_endpoint` class template describes an endpoint that may be associated with a particular UNIX socket.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/local/basic_endpoint.hpp

Convenience header: asio.hpp

5.157.1 local::basic_endpoint::basic_endpoint

Default constructor.

```
basic_endpoint();
```

Construct an endpoint using the specified path name.

```
basic_endpoint(
    const char * path_name);

basic_endpoint(
    const std::string & path_name);
```

Copy constructor.

```
basic_endpoint(
    const basic_endpoint & other);
```

5.157.1.1 local::basic_endpoint::basic_endpoint (1 of 4 overloads)

Default constructor.

```
basic_endpoint();
```

5.157.1.2 local::basic_endpoint::basic_endpoint (2 of 4 overloads)

Construct an endpoint using the specified path name.

```
basic_endpoint(
    const char * path_name);
```

5.157.1.3 local::basic_endpoint::basic_endpoint (3 of 4 overloads)

Construct an endpoint using the specified path name.

```
basic_endpoint(
    const std::string & path_name);
```

5.157.1.4 local::basic_endpoint::basic_endpoint (4 of 4 overloads)

Copy constructor.

```
basic_endpoint(
    const basic_endpoint & other);
```

5.157.2 local::basic_endpoint::capacity

Get the capacity of the endpoint in the native type.

```
std::size_t capacity() const;
```

5.157.3 local::basic_endpoint::data

Get the underlying endpoint in the native type.

```
data_type * data();  
  
const data_type * data() const;
```

5.157.3.1 local::basic_endpoint::data (1 of 2 overloads)

Get the underlying endpoint in the native type.

```
data_type * data();
```

5.157.3.2 local::basic_endpoint::data (2 of 2 overloads)

Get the underlying endpoint in the native type.

```
const data_type * data() const;
```

5.157.4 local::basic_endpoint::data_type

The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.

```
typedef implementation_defined data_type;
```

Requirements

Header: asio/local/basic_endpoint.hpp

Convenience header: asio.hpp

5.157.5 local::basic_endpoint::operator!=

Compare two endpoints for inequality.

```
friend bool operator!=  
    const basic_endpoint< Protocol > & e1,  
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/local/basic_endpoint.hpp

Convenience header: asio.hpp

5.157.6 local::basic_endpoint::operator<

Compare endpoints for ordering.

```
friend bool operator<(
    const basic_endpoint< Protocol > & e1,
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/local/basic_endpoint.hpp

Convenience header: asio.hpp

5.157.7 local::basic_endpoint::operator<<

Output an endpoint as a string.

```
std::basic_ostream< Elem, Traits > & operator<<(
    std::basic_ostream< Elem, Traits > & os,
    const basic_endpoint< Protocol > & endpoint);
```

Used to output a human-readable string for a specified endpoint.

Parameters

os The output stream to which the string will be written.

endpoint The endpoint to be written.

Return Value

The output stream.

5.157.8 local::basic_endpoint::operator<=

Compare endpoints for ordering.

```
friend bool operator<=
    const basic_endpoint< Protocol > & e1,
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/local/basic_endpoint.hpp

Convenience header: asio.hpp

5.157.9 local::basic_endpoint::operator=

Assign from another endpoint.

```
basic_endpoint & operator=(
    const basic_endpoint & other);
```

5.157.10 local::basic_endpoint::operator==

Compare two endpoints for equality.

```
friend bool operator==(  
    const basic_endpoint< Protocol > & e1,  
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/local/basic_endpoint.hpp

Convenience header: asio.hpp

5.157.11 local::basic_endpoint::operator>

Compare endpoints for ordering.

```
friend bool operator>(  
    const basic_endpoint< Protocol > & e1,  
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/local/basic_endpoint.hpp

Convenience header: asio.hpp

5.157.12 local::basic_endpoint::operator>=

Compare endpoints for ordering.

```
friend bool operator>=(  
    const basic_endpoint< Protocol > & e1,  
    const basic_endpoint< Protocol > & e2);
```

Requirements

Header: asio/local/basic_endpoint.hpp

Convenience header: asio.hpp

5.157.13 local::basic_endpoint::path

Get the path associated with the endpoint.

```
std::string path() const;
```

Set the path associated with the endpoint.

```
void path(  
    const char * p);  
  
void path(  
    const std::string & p);
```

5.157.13.1 local::basic_endpoint::path (1 of 3 overloads)

Get the path associated with the endpoint.

```
std::string path() const;
```

5.157.13.2 local::basic_endpoint::path (2 of 3 overloads)

Set the path associated with the endpoint.

```
void path(  
    const char * p);
```

5.157.13.3 local::basic_endpoint::path (3 of 3 overloads)

Set the path associated with the endpoint.

```
void path(  
    const std::string & p);
```

5.157.14 local::basic_endpoint::protocol

The protocol associated with the endpoint.

```
protocol_type protocol() const;
```

5.157.15 local::basic_endpoint::protocol_type

The protocol type associated with the endpoint.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/local/basic_endpoint.hpp

Convenience header: asio.hpp

5.157.16 local::basic_endpoint::resize

Set the underlying size of the endpoint in the native type.

```
void resize(  
    std::size_t new_size);
```

5.157.17 local::basic_endpoint::size

Get the underlying size of the endpoint in the native type.

```
std::size_t size() const;
```

5.158 local::connect_pair

Create a pair of connected sockets.

```
template<
    typename Protocol,
    typename SocketService1,
    typename SocketService2>
void connect_pair(
    basic_socket< Protocol, SocketService1 > & socket1,
    basic_socket< Protocol, SocketService2 > & socket2);

template<
    typename Protocol,
    typename SocketService1,
    typename SocketService2>
asio::error_code connect_pair(
    basic_socket< Protocol, SocketService1 > & socket1,
    basic_socket< Protocol, SocketService2 > & socket2,
    asio::error_code & ec);
```

Requirements

Header: asio/local/connect_pair.hpp

Convenience header: asio.hpp

5.158.1 local::connect_pair (1 of 2 overloads)

Create a pair of connected sockets.

```
template<
    typename Protocol,
    typename SocketService1,
    typename SocketService2>
void connect_pair(
    basic_socket< Protocol, SocketService1 > & socket1,
    basic_socket< Protocol, SocketService2 > & socket2);
```

5.158.2 local::connect_pair (2 of 2 overloads)

Create a pair of connected sockets.

```
template<
    typename Protocol,
    typename SocketService1,
    typename SocketService2>
asio::error_code connect_pair(
    basic_socket< Protocol, SocketService1 > & socket1,
    basic_socket< Protocol, SocketService2 > & socket2,
    asio::error_code & ec);
```

5.159 local::datagram_protocol

Encapsulates the flags needed for datagram-oriented UNIX sockets.

```
class datagram_protocol
```

Types

Name	Description
endpoint	The type of a UNIX domain endpoint.
socket	The UNIX domain socket type.

Member Functions

Name	Description
family	Obtain an identifier for the protocol family.
protocol	Obtain an identifier for the protocol.
type	Obtain an identifier for the type of the protocol.

The `local::datagram_protocol` class contains flags necessary for datagram-oriented UNIX domain sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: `asio/local/datagram_protocol.hpp`

Convenience header: `asio.hpp`

5.159.1 local::datagram_protocol::endpoint

The type of a UNIX domain endpoint.

```
typedef basic_endpoint< datagram_protocol > endpoint;
```

Types

Name	Description
data_type	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
protocol_type	The protocol type associated with the endpoint.

Member Functions

Name	Description
basic_endpoint	Default constructor. Construct an endpoint using the specified path name. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
path	Get the path associated with the endpoint. Set the path associated with the endpoint.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

Related Functions

Name	Description
operator<<	Output an endpoint as a string.

The `local::basic_endpoint` class template describes an endpoint that may be associated with a particular UNIX socket.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/local/datagram_protocol.hpp`

Convenience header: `asio.hpp`

5.159.2 local::datagram_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.159.3 local::datagram_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.159.4 local::datagram_protocol::socket

The UNIX domain socket type.

```
typedef basic_datagram_socket< datagram_protocol > socket;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.

Name	Description
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive on a connected socket.
async_receive_from	Start an asynchronous receive.

Name	Description
async_send	Start an asynchronous send on a connected socket.
async_send_to	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_datagram_socket	Construct a basic_datagram_socket without opening it. Construct and open a basic_datagram_socket. Construct a basic_datagram_socket, opening it and binding it to the given local endpoint. Construct a basic_datagram_socket on an existing native socket. Move-construct a basic_datagram_socket from another. Move-construct a basic_datagram_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.

Name	Description
operator=	Move-assign a basic_datagram_socket from another. Move-assign a basic_datagram_socket from a socket of another protocol type.
receive	Receive some data on a connected socket.
receive_from	Receive a datagram with the endpoint of the sender.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on a connected socket.
send_to	Send a datagram to the specified endpoint.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_datagram_socket` class template provides asynchronous and blocking datagram-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/local/datagram_protocol.hpp`

Convenience header: `asio.hpp`

5.159.5 `local::datagram_protocol::type`

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.160 `local::stream_protocol`

Encapsulates the flags needed for stream-oriented UNIX sockets.

```
class stream_protocol
```

Types

Name	Description
acceptor	The UNIX domain acceptor type.
endpoint	The type of a UNIX domain endpoint.
iostream	The UNIX domain iostream type.
socket	The UNIX domain socket type.

Member Functions

Name	Description
family	Obtain an identifier for the protocol family.
protocol	Obtain an identifier for the protocol.
type	Obtain an identifier for the type of the protocol.

The `local::stream_protocol` class contains flags necessary for stream-oriented UNIX domain sockets.

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: `asio/local/stream_protocol.hpp`

Convenience header: `asio.hpp`

5.160.1 local::stream_protocol::acceptor

The UNIX domain acceptor type.

```
typedef basic_socket_acceptor< stream_protocol > acceptor;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.

Name	Description
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of an acceptor.
native_type	(Deprecated: Use native_handle_type.) The native representation of an acceptor.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
accept	Accept a new connection. Accept a new connection and obtain the endpoint of the peer.
assign	Assigns an existing native acceptor to the acceptor.
async_accept	Start an asynchronous accept.
basic_socket_acceptor	Construct an acceptor without opening it. Construct an open acceptor. Construct an acceptor opened on the given endpoint. Construct a basic_socket_acceptor on an existing native acceptor. Move-construct a basic_socket_acceptor from another. Move-construct a basic_socket_acceptor from an acceptor of another protocol type.
bind	Bind the acceptor to the given local endpoint.

Name	Description
cancel	Cancel all asynchronous operations associated with the acceptor.
close	Close the acceptor.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the acceptor.
io_control	Perform an IO control command on the acceptor.
is_open	Determine whether the acceptor is open.
listen	Place the acceptor into the state where it will listen for new connections.
local_endpoint	Get the local endpoint of the acceptor.
native	(Deprecated: Use native_handle().) Get the native acceptor representation.
native_handle	Get the native acceptor representation.
native_non_blocking	Gets the non-blocking mode of the native acceptor implementation. Sets the non-blocking mode of the native acceptor implementation.
non_blocking	Gets the non-blocking mode of the acceptor. Sets the non-blocking mode of the acceptor.
open	Open the acceptor using the specified protocol.
operator=	Move-assign a basic_socket_acceptor from another. Move-assign a basic_socket_acceptor from an acceptor of another protocol type.
set_option	Set an option on the acceptor.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_socket_acceptor` class template is used for accepting new socket connections.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Example

Opening a socket acceptor with the SO_REUSEADDR option enabled:

```
asio::ip::tcp::acceptor acceptor(io_service);
asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), port);
acceptor.open(endpoint.protocol());
acceptor.set_option(asio::ip::tcp::acceptor::reuse_address(true));
acceptor.bind(endpoint);
acceptor.listen();
```

Requirements

Header: `asio/local/stream_protocol.hpp`

Convenience header: `asio.hpp`

5.160.2 local::stream_protocol::endpoint

The type of a UNIX domain endpoint.

```
typedef basic_endpoint< stream_protocol > endpoint;
```

Types

Name	Description
data_type	The type of the endpoint structure. This type is dependent on the underlying implementation of the socket layer.
protocol_type	The protocol type associated with the endpoint.

Member Functions

Name	Description
basic_endpoint	Default constructor. Construct an endpoint using the specified path name. Copy constructor.
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint.
path	Get the path associated with the endpoint. Set the path associated with the endpoint.
protocol	The protocol associated with the endpoint.
resize	Set the underlying size of the endpoint in the native type.
size	Get the underlying size of the endpoint in the native type.

Friends

Name	Description
operator!=	Compare two endpoints for inequality.
operator<	Compare endpoints for ordering.
operator<=	Compare endpoints for ordering.
operator==	Compare two endpoints for equality.
operator>	Compare endpoints for ordering.
operator>=	Compare endpoints for ordering.

Related Functions

Name	Description
operator<<	Output an endpoint as a string.

The `local::basic_endpoint` class template describes an endpoint that may be associated with a particular UNIX socket.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/local/stream_protocol.hpp`

Convenience header: `asio.hpp`

5.160.3 local::stream_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.160.4 local::stream_protocol::iostream

The UNIX domain iostream type.

```
typedef basic_socket_iostream< stream_protocol > iostream;
```

Types

Name	Description
duration_type	The duration type.
endpoint_type	The endpoint type.
time_type	The time type.

Member Functions

Name	Description
basic_socket_iostream	Construct a <code>basic_socket_iostream</code> without establishing a connection. Establish a connection to an endpoint corresponding to a resolver query.

Name	Description
close	Close the connection.
connect	Establish a connection to an endpoint corresponding to a resolver query.
error	Get the last error associated with the stream.
expires_at	Get the stream's expiry time as an absolute time. Set the stream's expiry time as an absolute time.
expires_from_now	Get the timer's expiry time relative to now. Set the stream's expiry time relative to now.
rdbuf	Return a pointer to the underlying streambuf.

Requirements

Header: asio/local/stream_protocol.hpp

Convenience header: asio.hpp

5.160.5 local::stream_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.160.6 local::stream_protocol::socket

The UNIX domain socket type.

```
typedef basic_stream_socket< stream_protocol > socket;
```

Types

Name	Description
broadcast	Socket option to permit sending of broadcast messages.
bytes_readable	IO control command to get the amount of data that can be read without blocking.
debug	Socket option to enable socket-level debugging.
do_not_route	Socket option to prevent routing, use local interfaces only.
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.

Name	Description
implementation_type	The underlying implementation type of I/O object.
keep_alive	Socket option to send keep-alives.
linger	Socket option to specify whether the socket lingers on close if unsent data is present.
lowest_layer_type	A basic_socket is always the lowest layer.
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
native_type	(Deprecated: Use native_handle_type.) The native representation of a socket.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the socket.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.
receive_low_watermark	Socket option for the receive low watermark.
reuse_address	Socket option to allow the socket to be bound to an address that is already in use.
send_buffer_size	Socket option for the send buffer size of a socket.
send_low_watermark	Socket option for the send low watermark.
service_type	The type of the service that will be used to provide I/O operations.
shutdown_type	Different ways a socket may be shutdown.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_read_some	Start an asynchronous read.
async_receive	Start an asynchronous receive.
async_send	Start an asynchronous send.

Name	Description
async_write_some	Start an asynchronous write.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
basic_stream_socket	Construct a basic_stream_socket without opening it. Construct and open a basic_stream_socket. Construct a basic_stream_socket, opening it and binding it to the given local endpoint. Construct a basic_stream_socket on an existing native socket. Move-construct a basic_stream_socket from another. Move-construct a basic_stream_socket from a socket of another protocol type.
bind	Bind the socket to the given local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the socket.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint of the socket.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native socket representation.
native_handle	Get the native socket representation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open the socket using the specified protocol.

Name	Description
operator=	Move-assign a basic_stream_socket from another. Move-assign a basic_stream_socket from a socket of another protocol type.
read_some	Read some data from the socket.
receive	Receive some data on the socket. Receive some data on a connected socket.
remote_endpoint	Get the remote endpoint of the socket.
send	Send some data on the socket.
set_option	Set an option on the socket.
shutdown	Disable sends or receives on the socket.
write_some	Write some data to the socket.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_stream_socket` class template provides asynchronous and blocking stream-oriented socket functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/local/stream_protocol.hpp`

Convenience header: `asio.hpp`

5.160.7 local::stream_protocol::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.161 mutable_buffer

Holds a buffer that can be modified.

```
class mutable_buffer
```

Member Functions

Name	Description
mutable_buffer	Construct an empty buffer. Construct a buffer to represent a given memory range.

Related Functions

Name	Description
operator+	Create a new modifiable buffer that is offset from the start of another.

The `mutable_buffer` class provides a safe representation of a buffer that can be modified. It does not own the underlying data, and so is cheap to copy or assign.

Accessing Buffer Contents

The contents of a buffer may be accessed using the `buffer_size` and `buffer_cast` functions:

```
asio::mutable_buffer b1 = ...;
std::size_t s1 = asio::buffer_size(b1);
unsigned char* p1 = asio::buffer_cast<unsigned char*>(b1);
```

The `asio::buffer_cast` function permits violations of type safety, so uses of it in application code should be carefully considered.

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.161.1 `mutable_buffer::mutable_buffer`

Construct an empty buffer.

```
mutable_buffer();
```

Construct a buffer to represent a given memory range.

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

5.161.1.1 `mutable_buffer::mutable_buffer (1 of 2 overloads)`

Construct an empty buffer.

```
mutable_buffer();
```

5.161.1.2 `mutable_buffer::mutable_buffer (2 of 2 overloads)`

Construct a buffer to represent a given memory range.

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

5.161.2 `mutable_buffer::operator+`

Create a new modifiable buffer that is offset from the start of another.

```
mutable_buffer operator+
    const mutable_buffer & b,
    std::size_t start);

mutable_buffer operator+
    std::size_t start,
    const mutable_buffer & b);
```

5.161.2.1 `mutable_buffer::operator+ (1 of 2 overloads)`

Create a new modifiable buffer that is offset from the start of another.

```
mutable_buffer operator+
    const mutable_buffer & b,
    std::size_t start);
```

5.161.2.2 `mutable_buffer::operator+ (2 of 2 overloads)`

Create a new modifiable buffer that is offset from the start of another.

```
mutable_buffer operator+
    std::size_t start,
    const mutable_buffer & b);
```

5.162 `mutable_buffers_1`

Adapts a single modifiable buffer so that it meets the requirements of the `MutableBufferSequence` concept.

```
class mutable_buffers_1 :
    public mutable_buffer
```

Types

Name	Description
<code>const_iterator</code>	A random-access iterator type that may be used to read elements.
<code>value_type</code>	The type for each element in the list of buffers.

Member Functions

Name	Description
<code>begin</code>	Get a random-access iterator to the first element.

Name	Description
end	Get a random-access iterator for one past the last element.
mutable_buffers_1	Construct to represent a given memory range. Construct to represent a single modifiable buffer.

Related Functions

Name	Description
operator+	Create a new modifiable buffer that is offset from the start of another.

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.162.1 mutable_buffers_1::begin

Get a random-access iterator to the first element.

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

5.162.2 mutable_buffers_1::const_iterator

A random-access iterator type that may be used to read elements.

```
typedef const mutable_buffer * const_iterator;
```

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.162.3 mutable_buffers_1::end

Get a random-access iterator for one past the last element.

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

5.162.4 `mutable_buffers_1::mutable_buffers_1`

Construct to represent a given memory range.

```
mutable_buffers_1(
    void * data,
    std::size_t size);
```

Construct to represent a single modifiable buffer.

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

5.162.4.1 `mutable_buffers_1::mutable_buffers_1 (1 of 2 overloads)`

Construct to represent a given memory range.

```
mutable_buffers_1(
    void * data,
    std::size_t size);
```

5.162.4.2 `mutable_buffers_1::mutable_buffers_1 (2 of 2 overloads)`

Construct to represent a single modifiable buffer.

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

5.162.5 `mutable_buffers_1::operator+`

Create a new modifiable buffer that is offset from the start of another.

```
mutable_buffer operator+
    const mutable_buffer & b,
    std::size_t start);
```

```
mutable_buffer operator+
    std::size_t start,
    const mutable_buffer & b);
```

5.162.5.1 `mutable_buffers_1::operator+ (1 of 2 overloads)`

Inherited from `mutable_buffer`.

Create a new modifiable buffer that is offset from the start of another.

```
mutable_buffer operator+
    const mutable_buffer & b,
    std::size_t start);
```

5.162.5.2 `mutable_buffers_1::operator+` (2 of 2 overloads)

Inherited from `mutable_buffer`.

Create a new modifiable buffer that is offset from the start of another.

```
mutable_buffer operator+
    std::size_t start,
    const mutable_buffer & b);
```

5.162.6 `mutable_buffers_1::value_type`

The type for each element in the list of buffers.

```
typedef mutable_buffer value_type;
```

Member Functions

Name	Description
<code>mutable_buffer</code>	Construct an empty buffer. Construct a buffer to represent a given memory range.

Related Functions

Name	Description
<code>operator+</code>	Create a new modifiable buffer that is offset from the start of another.

The `mutable_buffer` class provides a safe representation of a buffer that can be modified. It does not own the underlying data, and so is cheap to copy or assign.

Accessing Buffer Contents

The contents of a buffer may be accessed using the `buffer_size` and `buffer_cast` functions:

```
asio::mutable_buffer b1 = ...;
std::size_t s1 = asio::buffer_size(b1);
unsigned char* p1 = asio::buffer_cast<unsigned char*>(b1);
```

The `asio::buffer_cast` function permits violations of type safety, so uses of it in application code should be carefully considered.

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.163 null_buffers

An implementation of both the ConstBufferSequence and MutableBufferSequence concepts to represent a null buffer sequence.

```
class null_buffers
```

Types

Name	Description
const_iterator	A random-access iterator type that may be used to read elements.
value_type	The type for each element in the list of buffers.

Member Functions

Name	Description
begin	Get a random-access iterator to the first element.
end	Get a random-access iterator for one past the last element.

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.163.1 null_buffers::begin

Get a random-access iterator to the first element.

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

5.163.2 null_buffers::const_iterator

A random-access iterator type that may be used to read elements.

```
typedef const mutable_buffer * const_iterator;
```

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.163.3 null_buffers::end

Get a random-access iterator for one past the last element.

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

5.163.4 null_buffers::value_type

The type for each element in the list of buffers.

```
typedef mutable_buffer value_type;
```

Member Functions

Name	Description
mutable_buffer	Construct an empty buffer. Construct a buffer to represent a given memory range.

Related Functions

Name	Description
operator+	Create a new modifiable buffer that is offset from the start of another.

The `mutable_buffer` class provides a safe representation of a buffer that can be modified. It does not own the underlying data, and so is cheap to copy or assign.

Accessing Buffer Contents

The contents of a buffer may be accessed using the `buffer_size` and `buffer_cast` functions:

```
asio::mutable_buffer b1 = ...;
std::size_t s1 = asio::buffer_size(b1);
unsigned char* p1 = asio::buffer_cast<unsigned char*>(b1);
```

The `asio::buffer_cast` function permits violations of type safety, so uses of it in application code should be carefully considered.

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.164 operator<<

Output an error code.

```
template<
    typename Elem,
    typename Traits>
std::basic_ostream<Elem, Traits> & operator<<(
    std::basic_ostream<Elem, Traits> & os,
    const error_code & ec);
```

Requirements

Header: asio/error_code.hpp

Convenience header: asio.hpp

5.165 placeholders::bytes_transferred

An argument placeholder, for use with boost::bind(), that corresponds to the bytes_transferred argument of a handler for asynchronous functions such as `asio::basic_stream_socket::async_write_some` or `asio::async_write`.

```
unspecified bytes_transferred;
```

Requirements

Header: asio/placeholders.hpp

Convenience header: asio.hpp

5.166 placeholders::error

An argument placeholder, for use with boost::bind(), that corresponds to the error argument of a handler for any of the asynchronous functions.

```
unspecified error;
```

Requirements

Header: asio/placeholders.hpp

Convenience header: asio.hpp

5.167 placeholders::iterator

An argument placeholder, for use with boost::bind(), that corresponds to the iterator argument of a handler for asynchronous functions such as `asio::basic_resolver::async_resolve`.

```
unspecified iterator;
```

Requirements

Header: asio/placeholders.hpp

Convenience header: asio.hpp

5.168 placeholders::signal_number

An argument placeholder, for use with `boost::bind()`, that corresponds to the `signal_number` argument of a handler for asynchronous functions such as `asio::signal_set::async_wait`.

```
unspecified signal_number;
```

Requirements

Header: `asio/placeholders.hpp`

Convenience header: `asio.hpp`

5.169 posix::basic_descriptor

Provides POSIX descriptor functionality.

```
template<
    typename DescriptorService>
class basic_descriptor :
    public basic_io_object<DescriptorService>,
    public posix::descriptor_base
```

Types

Name	Description
bytes_readable	IO control command to get the amount of data that can be read without blocking.
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A <code>basic_descriptor</code> is always the lowest layer.
native_handle_type	The native representation of a descriptor.
native_type	(Deprecated: Use <code>native_handle_type</code> .) The native representation of a descriptor.
non_blocking_io	(Deprecated: Use <code>non_blocking()</code> .) IO control command to set the blocking mode of the descriptor.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native descriptor to the descriptor.

Name	Description
basic_descriptor	Construct a basic_descriptor without opening it. Construct a basic_descriptor on an existing native descriptor. Move-construct a basic_descriptor from another.
cancel	Cancel all asynchronous operations associated with the descriptor.
close	Close the descriptor.
get_io_service	Get the io_service associated with the object.
io_control	Perform an IO control command on the descriptor.
is_open	Determine whether the descriptor is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native descriptor representation.
native_handle	Get the native descriptor representation.
native_non_blocking	Gets the non-blocking mode of the native descriptor implementation. Sets the non-blocking mode of the native descriptor implementation.
non_blocking	Gets the non-blocking mode of the descriptor. Sets the non-blocking mode of the descriptor.
operator=	Move-assign a basic_descriptor from another.
release	Release ownership of the native descriptor implementation.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_descriptor	Protected destructor to prevent deletion through this type.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `posix::basic_descriptor` class template provides the ability to wrap a POSIX descriptor.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/posix/basic_descriptor.hpp`

Convenience header: `asio.hpp`

5.169.1 `posix::basic_descriptor::assign`

Assign an existing native descriptor to the descriptor.

```
void assign(
    const native_handle_type & native_descriptor);

asio::error_code assign(
    const native_handle_type & native_descriptor,
    asio::error_code & ec);
```

5.169.1.1 `posix::basic_descriptor::assign (1 of 2 overloads)`

Assign an existing native descriptor to the descriptor.

```
void assign(
    const native_handle_type & native_descriptor);
```

5.169.1.2 `posix::basic_descriptor::assign (2 of 2 overloads)`

Assign an existing native descriptor to the descriptor.

```
asio::error_code assign(
    const native_handle_type & native_descriptor,
    asio::error_code & ec);
```

5.169.2 posix::basic_descriptor::basic_descriptor

Construct a `posix::basic_descriptor` without opening it.

```
explicit basic_descriptor(
    asio::io_service & io_service);
```

Construct a `posix::basic_descriptor` on an existing native descriptor.

```
basic_descriptor(
    asio::io_service & io_service,
    const native_handle_type & native_descriptor);
```

Move-construct a `posix::basic_descriptor` from another.

```
basic_descriptor(
    basic_descriptor && other);
```

5.169.2.1 posix::basic_descriptor::basic_descriptor (1 of 3 overloads)

Construct a `posix::basic_descriptor` without opening it.

```
basic_descriptor(
    asio::io_service & io_service);
```

This constructor creates a descriptor without opening it.

Parameters

io_service The `io_service` object that the descriptor will use to dispatch handlers for any asynchronous operations performed on the descriptor.

5.169.2.2 posix::basic_descriptor::basic_descriptor (2 of 3 overloads)

Construct a `posix::basic_descriptor` on an existing native descriptor.

```
basic_descriptor(
    asio::io_service & io_service,
    const native_handle_type & native_descriptor);
```

This constructor creates a descriptor object to hold an existing native descriptor.

Parameters

io_service The `io_service` object that the descriptor will use to dispatch handlers for any asynchronous operations performed on the descriptor.

native_descriptor A native descriptor.

Exceptions

`asio::system_error` Thrown on failure.

5.169.2.3 `posix::basic_descriptor::basic_descriptor` (3 of 3 overloads)

Move-construct a `posix::basic_descriptor` from another.

```
basic_descriptor(
    basic_descriptor && other);
```

This constructor moves a descriptor from one object to another.

Parameters

other The other `posix::basic_descriptor` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_descriptor(io_service&)` constructor.

5.169.3 `posix::basic_descriptor::bytes_readable`

Inherited from `posix::descriptor_base`.

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::posix::stream_descriptor descriptor(io_service);
...
asio::descriptor_base::bytes_readable command(true);
descriptor.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: `asio/posix/basic_descriptor.hpp`

Convenience header: `asio.hpp`

5.169.4 `posix::basic_descriptor::cancel`

Cancel all asynchronous operations associated with the descriptor.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.169.4.1 posix::basic_descriptor::cancel (1 of 2 overloads)

Cancel all asynchronous operations associated with the descriptor.

```
void cancel();
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.169.4.2 posix::basic_descriptor::cancel (2 of 2 overloads)

Cancel all asynchronous operations associated with the descriptor.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

`ec` Set to indicate what error occurred, if any.

5.169.5 posix::basic_descriptor::close

Close the descriptor.

```
void close();

asio::error_code close(
    asio::error_code & ec);
```

5.169.5.1 posix::basic_descriptor::close (1 of 2 overloads)

Close the descriptor.

```
void close();
```

This function is used to close the descriptor. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure. Note that, even if the function indicates an error, the underlying descriptor is closed.

5.169.5.2 posix::basic_descriptor::close (2 of 2 overloads)

Close the descriptor.

```
asio::error_code close(  
    asio::error_code & ec);
```

This function is used to close the descriptor. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any. Note that, even if the function indicates an error, the underlying descriptor is closed.

5.169.6 posix::basic_descriptor::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.169.6.1 posix::basic_descriptor::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.169.6.2 posix::basic_descriptor::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.169.7 posix::basic_descriptor::get_io_service

Inherited from basic_io_object.

Get the `io_service` associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the `io_service` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_service` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.169.8 posix::basic_descriptor::get_service

Get the service associated with the I/O object.

```
service_type & get_service();  
  
const service_type & get_service() const;
```

5.169.8.1 posix::basic_descriptor::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.169.8.2 posix::basic_descriptor::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.169.9 posix::basic_descriptor::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.169.10 posix::basic_descriptor::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asioposix/basic_descriptor.hpp

Convenience header: asio.hpp

5.169.11 posix::basic_descriptor::io_control

Perform an IO control command on the descriptor.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.169.11.1 posix::basic_descriptor::io_control (1 of 2 overloads)

Perform an IO control command on the descriptor.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);
```

This function is used to execute an IO control command on the descriptor.

Parameters

command The IO control command to be performed on the descriptor.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the number of bytes ready to read:

```
asio::posix::stream_descriptor descriptor(io_service);
...
asio::posix::stream_descriptor::bytes_readable command;
descriptor.io_control(command);
std::size_t bytes_readable = command.get();
```

5.169.11.2 posix::basic_descriptor::io_control (2 of 2 overloads)

Perform an IO control command on the descriptor.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

This function is used to execute an IO control command on the descriptor.

Parameters

command The IO control command to be performed on the descriptor.

ec Set to indicate what error occurred, if any.

Example

Getting the number of bytes ready to read:

```
asio::posix::stream_descriptor descriptor(io_service);
...
asio::posix::stream_descriptor::bytes_readable command;
asio::error_code ec;
descriptor.io_control(command, ec);
if (ec)
{
    // An error occurred.
}
std::size_t bytes_readable = command.get();
```

5.169.12 posix::basic_descriptor::is_open

Determine whether the descriptor is open.

```
bool is_open() const;
```

5.169.13 posix::basic_descriptor::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.169.13.1 posix::basic_descriptor::lowest_layer (1 of 2 overloads)

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a `posix::basic_descriptor` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.169.13.2 `posix::basic_descriptor::lowest_layer` (2 of 2 overloads)

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a `posix::basic_descriptor` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.169.14 `posix::basic_descriptor::lowest_layer_type`

A `posix::basic_descriptor` is always the lowest layer.

```
typedef basic_descriptor< DescriptorService > lowest_layer_type;
```

Types

Name	Description
<code>bytes_readable</code>	IO control command to get the amount of data that can be read without blocking.
<code>implementation_type</code>	The underlying implementation type of I/O object.
<code>lowest_layer_type</code>	A <code>basic_descriptor</code> is always the lowest layer.
<code>native_handle_type</code>	The native representation of a descriptor.
<code>native_type</code>	(Deprecated: Use <code>native_handle_type</code> .) The native representation of a descriptor.
<code>non_blocking_io</code>	(Deprecated: Use <code>non_blocking()</code> .) IO control command to set the blocking mode of the descriptor.
<code>service_type</code>	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
<code>assign</code>	Assign an existing native descriptor to the descriptor.
<code>basic_descriptor</code>	Construct a <code>basic_descriptor</code> without opening it. Construct a <code>basic_descriptor</code> on an existing native descriptor. Move-construct a <code>basic_descriptor</code> from another.

Name	Description
cancel	Cancel all asynchronous operations associated with the descriptor.
close	Close the descriptor.
get_io_service	Get the io_service associated with the object.
io_control	Perform an IO control command on the descriptor.
is_open	Determine whether the descriptor is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native descriptor representation.
native_handle	Get the native descriptor representation.
native_non_blocking	Gets the non-blocking mode of the native descriptor implementation. Sets the non-blocking mode of the native descriptor implementation.
non_blocking	Gets the non-blocking mode of the descriptor. Sets the non-blocking mode of the descriptor.
operator=	Move-assign a basic_descriptor from another.
release	Release ownership of the native descriptor implementation.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_descriptor	Protected destructor to prevent deletion through this type.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `posix::basic_descriptor` class template provides the ability to wrap a POSIX descriptor.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio posix/basic_descriptor.hpp`

Convenience header: `asio.hpp`

5.169.15 `posix::basic_descriptor::native`

(Deprecated: Use `native_handle()`) Get the native descriptor representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the descriptor. This is intended to allow access to native descriptor functionality that is not otherwise provided.

5.169.16 `posix::basic_descriptor::native_handle`

Get the native descriptor representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the descriptor. This is intended to allow access to native descriptor functionality that is not otherwise provided.

5.169.17 `posix::basic_descriptor::native_handle_type`

The native representation of a descriptor.

```
typedef DescriptorService::native_handle_type native_handle_type;
```

Requirements

Header: `asio posix/basic_descriptor.hpp`

Convenience header: `asio.hpp`

5.169.18 `posix::basic_descriptor::native_non_blocking`

Gets the non-blocking mode of the native descriptor implementation.

```
bool native_non_blocking() const;
```

Sets the non-blocking mode of the native descriptor implementation.

```
void native_non_blocking(
    bool mode);

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.169.18.1 posix::basic_descriptor::native_non_blocking (1 of 3 overloads)

Gets the non-blocking mode of the native descriptor implementation.

```
bool native_non_blocking() const;
```

This function is used to retrieve the non-blocking mode of the underlying native descriptor. This mode has no effect on the behaviour of the descriptor object's synchronous operations.

Return Value

`true` if the underlying descriptor is in non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Remarks

The current non-blocking mode is cached by the descriptor object. Consequently, the return value may be incorrect if the non-blocking mode was set directly on the native descriptor.

5.169.18.2 posix::basic_descriptor::native_non_blocking (2 of 3 overloads)

Sets the non-blocking mode of the native descriptor implementation.

```
void native_non_blocking(
    bool mode);
```

This function is used to modify the non-blocking mode of the underlying native descriptor. It has no effect on the behaviour of the descriptor object's synchronous operations.

Parameters

mode If `true`, the underlying descriptor is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Exceptions

asio::system_error Thrown on failure. If the mode is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

5.169.18.3 posix::basic_descriptor::native_non_blocking (3 of 3 overloads)

Sets the non-blocking mode of the native descriptor implementation.

```
asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

This function is used to modify the non-blocking mode of the underlying native descriptor. It has no effect on the behaviour of the descriptor object's synchronous operations.

Parameters

mode If `true`, the underlying descriptor is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

ec Set to indicate what error occurred, if any. If the mode is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

5.169.19 `posix::basic_descriptor::native_type`

(Deprecated: Use `native_handle_type`.) The native representation of a descriptor.

```
typedef DescriptorService::native_handle_type native_type;
```

Requirements

Header: `asio posix/basic_descriptor.hpp`

Convenience header: `asio.hpp`

5.169.20 `posix::basic_descriptor::non_blocking`

Gets the non-blocking mode of the descriptor.

```
bool non_blocking() const;
```

Sets the non-blocking mode of the descriptor.

```
void non_blocking(
    bool mode);

asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.169.20.1 `posix::basic_descriptor::non_blocking (1 of 3 overloads)`

Gets the non-blocking mode of the descriptor.

```
bool non_blocking() const;
```

Return Value

`true` if the descriptor's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.169.20.2 `posix::basic_descriptor::non_blocking` (2 of 3 overloads)

Sets the non-blocking mode of the descriptor.

```
void non_blocking(
    bool mode);
```

Parameters

mode If `true`, the descriptor's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.169.20.3 `posix::basic_descriptor::non_blocking` (3 of 3 overloads)

Sets the non-blocking mode of the descriptor.

```
asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

Parameters

mode If `true`, the descriptor's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

ec Set to indicate what error occurred, if any.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.169.21 `posix::basic_descriptor::non_blocking_io`

Inherited from `posix::descriptor_base`.

(Deprecated: Use `non_blocking()`) IO control command to set the blocking mode of the descriptor.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::posix::stream_descriptor descriptor(io_service);
...
asio::descriptor_base::non_blocking_io command(true);
descriptor.io_control(command);
```

Requirements

Header: asio posix/basic_descriptor.hpp

Convenience header: asio.hpp

5.169.22 posix::basic_descriptor::operator=

Move-assign a `posix::basic_descriptor` from another.

```
basic_descriptor & operator=(  
    basic_descriptor && other);
```

This assignment operator moves a descriptor from one object to another.

Parameters

other The other `posix::basic_descriptor` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_descriptor(io_service&)` constructor.

5.169.23 posix::basic_descriptor::release

Release ownership of the native descriptor implementation.

```
native_handle_type release();
```

This function may be used to obtain the underlying representation of the descriptor. After calling this function, `is_open()` returns false. The caller is responsible for closing the descriptor.

All outstanding asynchronous read or write operations will finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

5.169.24 posix::basic_descriptor::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`.) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.169.25 posix::basic_descriptor::service_type

Inherited from `basic_io_object`.

The type of the service that will be used to provide I/O operations.

```
typedef DescriptorService service_type;
```

Requirements

Header: `asio/posix/basic_descriptor.hpp`

Convenience header: `asio.hpp`

5.169.26 posix::basic_descriptor::~basic_descriptor

Protected destructor to prevent deletion through this type.

```
~basic_descriptor();
```

5.170 posix::basic_stream_descriptor

Provides stream-oriented descriptor functionality.

```
template<
    typename StreamDescriptorService = stream_descriptor_service>
class basic_stream_descriptor :
    public posix::basic_descriptor< StreamDescriptorService >
```

Types

Name	Description
<code>bytes_readable</code>	IO control command to get the amount of data that can be read without blocking.
<code>implementation_type</code>	The underlying implementation type of I/O object.
<code>lowest_layer_type</code>	A <code>basic_descriptor</code> is always the lowest layer.
<code>native_handle_type</code>	The native representation of a descriptor.
<code>native_type</code>	(Deprecated: Use <code>native_handle_type</code> .) The native representation of a descriptor.
<code>non_blocking_io</code>	(Deprecated: Use <code>non_blocking()</code> .) IO control command to set the blocking mode of the descriptor.
<code>service_type</code>	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native descriptor to the descriptor.
async_read_some	Start an asynchronous read.
async_write_some	Start an asynchronous write.
basic_stream_descriptor	Construct a basic_stream_descriptor without opening it. Construct a basic_stream_descriptor on an existing native descriptor. Move-construct a basic_stream_descriptor from another.
cancel	Cancel all asynchronous operations associated with the descriptor.
close	Close the descriptor.
get_io_service	Get the io_service associated with the object.
io_control	Perform an IO control command on the descriptor.
is_open	Determine whether the descriptor is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native descriptor representation.
native_handle	Get the native descriptor representation.
native_non_blocking	Gets the non-blocking mode of the native descriptor implementation. Sets the non-blocking mode of the native descriptor implementation.
non_blocking	Gets the non-blocking mode of the descriptor. Sets the non-blocking mode of the descriptor.
operator=	Move-assign a basic_stream_descriptor from another.
read_some	Read some data from the descriptor.
release	Release ownership of the native descriptor implementation.
write_some	Write some data to the descriptor.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.

Name	Description
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `posix::basic_stream_descriptor` class template provides asynchronous and blocking stream-oriented descriptor functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio posix/basic_stream_descriptor.hpp`

Convenience header: `asio.hpp`

5.170.1 `posix::basic_stream_descriptor::assign`

Assign an existing native descriptor to the descriptor.

```
void assign(
    const native_handle_type & native_descriptor);

asio::error_code assign(
    const native_handle_type & native_descriptor,
    asio::error_code & ec);
```

5.170.1.1 `posix::basic_stream_descriptor::assign (1 of 2 overloads)`

Inherited from `posix::basic_descriptor`.

Assign an existing native descriptor to the descriptor.

```
void assign(
    const native_handle_type & native_descriptor);
```

5.170.1.2 `posix::basic_stream_descriptor::assign` (2 of 2 overloads)

Inherited from `posix::basic_descriptor`.

Assign an existing native descriptor to the descriptor.

```
asio::error_code assign(
    const native_handle_type & native_descriptor,
    asio::error_code & ec);
```

5.170.2 `posix::basic_stream_descriptor::async_read_some`

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

This function is used to asynchronously read data from the stream descriptor. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be read. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes read.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The read operation may not read all of the requested number of bytes. Consider using the `async_read` function if you need to ensure that the requested amount of data is read before the asynchronous operation completes.

Example

To read into a single data buffer use the `buffer` function as follows:

```
descriptor.async_read_some(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.170.3 posix::basic_stream_descriptor::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some(
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

This function is used to asynchronously write data to the stream descriptor. The function call always returns immediately.

Parameters

buffers One or more data buffers to be written to the descriptor. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes written.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The write operation may not transmit all of the data to the peer. Consider using the `async_write` function if you need to ensure that all data is written before the asynchronous operation completes.

Example

To write a single data buffer use the `buffer` function as follows:

```
descriptor.async_write_some(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.170.4 posix::basic_stream_descriptor::basic_stream_descriptor

Construct a `posix::basic_stream_descriptor` without opening it.

```
explicit basic_stream_descriptor(
    asio::io_service & io_service);
```

Construct a `posix::basic_stream_descriptor` on an existing native descriptor.

```
basic_stream_descriptor(
    asio::io_service & io_service,
    const native_handle_type & native_descriptor);
```

Move-construct a `posix::basic_stream_descriptor` from another.

```
basic_stream_descriptor(
    basic_stream_descriptor && other);
```

5.170.4.1 `posix::basic_stream_descriptor::basic_stream_descriptor (1 of 3 overloads)`

Construct a `posix::basic_stream_descriptor` without opening it.

```
basic_stream_descriptor(
    asio::io_service & io_service);
```

This constructor creates a stream descriptor without opening it. The descriptor needs to be opened and then connected or accepted before data can be sent or received on it.

Parameters

io_service The `io_service` object that the stream descriptor will use to dispatch handlers for any asynchronous operations performed on the descriptor.

5.170.4.2 `posix::basic_stream_descriptor::basic_stream_descriptor (2 of 3 overloads)`

Construct a `posix::basic_stream_descriptor` on an existing native descriptor.

```
basic_stream_descriptor(
    asio::io_service & io_service,
    const native_handle_type & native_descriptor);
```

This constructor creates a stream descriptor object to hold an existing native descriptor.

Parameters

io_service The `io_service` object that the stream descriptor will use to dispatch handlers for any asynchronous operations performed on the descriptor.

native_descriptor The new underlying descriptor implementation.

Exceptions

`asio::system_error` Thrown on failure.

5.170.4.3 `posix::basic_stream_descriptor::basic_stream_descriptor (3 of 3 overloads)`

Move-construct a `posix::basic_stream_descriptor` from another.

```
basic_stream_descriptor(
    basic_stream_descriptor && other);
```

This constructor moves a stream descriptor from one object to another.

Parameters

other The other `posix::basic_stream_descriptor` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_stream_descriptor(io_service&)` constructor.

5.170.5 `posix::basic_stream_descriptor::bytes_readable`

Inherited from `posix::descriptor_base`.

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::posix::stream_descriptor descriptor(io_service);
...
asio::descriptor_base::bytes_readable command(true);
descriptor.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: `asio posix/basic_stream_descriptor.hpp`

Convenience header: `asio.hpp`

5.170.6 `posix::basic_stream_descriptor::cancel`

Cancel all asynchronous operations associated with the descriptor.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.170.6.1 `posix::basic_stream_descriptor::cancel (1 of 2 overloads)`

Inherited from `posix::basic_descriptor`.

Cancel all asynchronous operations associated with the descriptor.

```
void cancel();
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

asio::system_error Thrown on failure.

5.170.6.2 posix::basic_stream_descriptor::cancel (2 of 2 overloads)

Inherited from posix::basic_descriptor.

Cancel all asynchronous operations associated with the descriptor.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

5.170.7 posix::basic_stream_descriptor::close

Close the descriptor.

```
void close();

asio::error_code close(
    asio::error_code & ec);
```

5.170.7.1 posix::basic_stream_descriptor::close (1 of 2 overloads)

Inherited from posix::basic_descriptor.

Close the descriptor.

```
void close();
```

This function is used to close the descriptor. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

asio::system_error Thrown on failure. Note that, even if the function indicates an error, the underlying descriptor is closed.

5.170.7.2 posix::basic_stream_descriptor::close (2 of 2 overloads)

Inherited from posix::basic_descriptor.

Close the descriptor.

```
asio::error_code close(
    asio::error_code & ec);
```

This function is used to close the descriptor. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any. Note that, even if the function indicates an error, the underlying descriptor is closed.

5.170.8 posix::basic_stream_descriptor::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.170.8.1 posix::basic_stream_descriptor::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.170.8.2 posix::basic_stream_descriptor::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.170.9 posix::basic_stream_descriptor::get_io_service

Inherited from basic_io_object.

Get the [io_service](#) associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the [io_service](#) object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the [io_service](#) object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.170.10 posix::basic_stream_descriptor::get_service

Get the service associated with the I/O object.

```
service_type & get_service();  
  
const service_type & get_service() const;
```

5.170.10.1 posix::basic_stream_descriptor::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.170.10.2 posix::basic_stream_descriptor::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.170.11 posix::basic_stream_descriptor::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.170.12 posix::basic_stream_descriptor::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio posix/basic_stream_descriptor.hpp

Convenience header: asio.hpp

5.170.13 posix::basic_stream_descriptor::io_control

Perform an IO control command on the descriptor.

```
void io_control(
    IoControlCommand & command);

asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.170.13.1 posix::basic_stream_descriptor::io_control (1 of 2 overloads)

Inherited from posix::basic_descriptor.

Perform an IO control command on the descriptor.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);
```

This function is used to execute an IO control command on the descriptor.

Parameters

command The IO control command to be performed on the descriptor.

Exceptions

asio::system_error Thrown on failure.

Example

Getting the number of bytes ready to read:

```
asio::posix::stream_descriptor descriptor(io_service);
...
asio::posix::stream_descriptor::bytes_readable command;
descriptor.io_control(command);
std::size_t bytes_readable = command.get();
```

5.170.13.2 posix::basic_stream_descriptor::io_control (2 of 2 overloads)

Inherited from posix::basic_descriptor.

Perform an IO control command on the descriptor.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

This function is used to execute an IO control command on the descriptor.

Parameters

command The IO control command to be performed on the descriptor.

ec Set to indicate what error occurred, if any.

Example

Getting the number of bytes ready to read:

```
asio::posix::stream_descriptor descriptor(io_service);
...
asio::posix::stream_descriptor::bytes_readable command;
asio::error_code ec;
descriptor.io_control(command, ec);
if (ec)
{
    // An error occurred.
}
std::size_t bytes_readable = command.get();
```

5.170.14 posix::basic_stream_descriptor::is_open

Inherited from posix::basic_descriptor.

Determine whether the descriptor is open.

```
bool is_open() const;
```

5.170.15 posix::basic_stream_descriptor::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.170.15.1 posix::basic_stream_descriptor::lowest_layer (1 of 2 overloads)

Inherited from posix::basic_descriptor.

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a [posix::basic_descriptor](#) cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.170.15.2 posix::basic_stream_descriptor::lowest_layer (2 of 2 overloads)

Inherited from posix::basic_descriptor.

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a [posix::basic_descriptor](#) cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.170.16 posix::basic_stream_descriptor::lowest_layer_type

Inherited from posix::basic_descriptor.

A [posix::basic_descriptor](#) is always the lowest layer.

```
typedef basic_descriptor< StreamDescriptorService > lowest_layer_type;
```

Types

Name	Description
bytes_readable	IO control command to get the amount of data that can be read without blocking.
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_descriptor is always the lowest layer.
native_handle_type	The native representation of a descriptor.
native_type	(Deprecated: Use native_handle_type.) The native representation of a descriptor.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the descriptor.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native descriptor to the descriptor.
basic_descriptor	Construct a basic_descriptor without opening it. Construct a basic_descriptor on an existing native descriptor. Move-construct a basic_descriptor from another.
cancel	Cancel all asynchronous operations associated with the descriptor.
close	Close the descriptor.
get_io_service	Get the io_service associated with the object.
io_control	Perform an IO control command on the descriptor.
is_open	Determine whether the descriptor is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native descriptor representation.
native_handle	Get the native descriptor representation.
native_non_blocking	Gets the non-blocking mode of the native descriptor implementation. Sets the non-blocking mode of the native descriptor implementation.

Name	Description
non_blocking	Gets the non-blocking mode of the descriptor. Sets the non-blocking mode of the descriptor.
operator=	Move-assign a basic_descriptor from another.
release	Release ownership of the native descriptor implementation.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_descriptor	Protected destructor to prevent deletion through this type.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `posix::basic_descriptor` class template provides the ability to wrap a POSIX descriptor.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio posix/basic_stream_descriptor.hpp`

Convenience header: `asio.hpp`

5.170.17 posix::basic_stream_descriptor::native

Inherited from `posix::basic_descriptor`.

(Deprecated: Use `native_handle()`.) Get the native descriptor representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the descriptor. This is intended to allow access to native descriptor functionality that is not otherwise provided.

5.170.18 posix::basic_stream_descriptor::native_handle

Inherited from posix::basic_descriptor.

Get the native descriptor representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the descriptor. This is intended to allow access to native descriptor functionality that is not otherwise provided.

5.170.19 posix::basic_stream_descriptor::native_handle_type

The native representation of a descriptor.

```
typedef StreamDescriptorService::native_handle_type native_handle_type;
```

Requirements

Header: asioposix/basic_stream_descriptor.hpp

Convenience header: asio.hpp

5.170.20 posix::basic_stream_descriptor::native_non_blocking

Gets the non-blocking mode of the native descriptor implementation.

```
bool native_non_blocking() const;
```

Sets the non-blocking mode of the native descriptor implementation.

```
void native_non_blocking(
    bool mode);

asio::error_code native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.170.20.1 posix::basic_stream_descriptor::native_non_blocking (1 of 3 overloads)

Inherited from posix::basic_descriptor.

Gets the non-blocking mode of the native descriptor implementation.

```
bool native_non_blocking() const;
```

This function is used to retrieve the non-blocking mode of the underlying native descriptor. This mode has no effect on the behaviour of the descriptor object's synchronous operations.

Return Value

true if the underlying descriptor is in non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Remarks

The current non-blocking mode is cached by the descriptor object. Consequently, the return value may be incorrect if the non-blocking mode was set directly on the native descriptor.

5.170.20.2 `posix::basic_stream_descriptor::native_non_blocking` (2 of 3 overloads)

Inherited from posix::basic_descriptor.

Sets the non-blocking mode of the native descriptor implementation.

```
void native_non_blocking(  
    bool mode);
```

This function is used to modify the non-blocking mode of the underlying native descriptor. It has no effect on the behaviour of the descriptor object's synchronous operations.

Parameters

mode If `true`, the underlying descriptor is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

Exceptions

asio::system_error Thrown on failure. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

5.170.20.3 `posix::basic_stream_descriptor::native_non_blocking` (3 of 3 overloads)

Inherited from posix::basic_descriptor.

Sets the non-blocking mode of the native descriptor implementation.

```
asio::error_code native_non_blocking(  
    bool mode,  
    asio::error_code & ec);
```

This function is used to modify the non-blocking mode of the underlying native descriptor. It has no effect on the behaviour of the descriptor object's synchronous operations.

Parameters

mode If `true`, the underlying descriptor is put into non-blocking mode and direct system calls may fail with `asio::error::would_block` (or the equivalent system error).

ec Set to indicate what error occurred, if any. If the `mode` is `false`, but the current value of `non_blocking()` is `true`, this function fails with `asio::error::invalid_argument`, as the combination does not make sense.

5.170.21 `posix::basic_stream_descriptor::native_type`

(Deprecated: Use `native_handle_type`.) The native representation of a descriptor.

```
typedef StreamDescriptorService::native_handle_type native_type;
```

Requirements

Header: asio posix/basic_stream_descriptor.hpp

Convenience header: asio.hpp

5.170.22 posix::basic_stream_descriptor::non_blocking

Gets the non-blocking mode of the descriptor.

```
bool non_blocking() const;
```

Sets the non-blocking mode of the descriptor.

```
void non_blocking(
    bool mode);

asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.170.22.1 posix::basic_stream_descriptor::non_blocking (1 of 3 overloads)

Inherited from posix::basic_descriptor.

Gets the non-blocking mode of the descriptor.

```
bool non_blocking() const;
```

Return Value

true if the descriptor's synchronous operations will fail with asio::error::would_block if they are unable to perform the requested operation immediately. If false, synchronous operations will block until complete.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error asio::error::would_block.

5.170.22.2 posix::basic_stream_descriptor::non_blocking (2 of 3 overloads)

Inherited from posix::basic_descriptor.

Sets the non-blocking mode of the descriptor.

```
void non_blocking(
    bool mode);
```

Parameters

mode If true, the descriptor's synchronous operations will fail with asio::error::would_block if they are unable to perform the requested operation immediately. If false, synchronous operations will block until complete.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.170.22.3 `posix::basic_stream_descriptor::non_blocking` (3 of 3 overloads)

Inherited from `posix::basic_descriptor`.

Sets the non-blocking mode of the descriptor.

```
asio::error_code non_blocking(
    bool mode,
    asio::error_code & ec);
```

Parameters

mode If `true`, the descriptor's synchronous operations will fail with `asio::error::would_block` if they are unable to perform the requested operation immediately. If `false`, synchronous operations will block until complete.

ec Set to indicate what error occurred, if any.

Remarks

The non-blocking mode has no effect on the behaviour of asynchronous operations. Asynchronous operations will never fail with the error `asio::error::would_block`.

5.170.23 `posix::basic_stream_descriptor::non_blocking_io`

Inherited from `posix::descriptor_base`.

(Deprecated: Use `non_blocking()`) IO control command to set the blocking mode of the descriptor.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::posix::stream_descriptor descriptor(io_service);
...
asio::descriptor_base::non_blocking_io command(true);
descriptor.io_control(command);
```

Requirements

Header: `asio posix/basic_stream_descriptor.hpp`

Convenience header: `asio.hpp`

5.170.24 posix::basic_stream_descriptor::operator=

Move-assign a `posix::basic_stream_descriptor` from another.

```
basic_stream_descriptor & operator=(  
    basic_stream_descriptor && other);
```

This assignment operator moves a stream descriptor from one object to another.

Parameters

other The other `posix::basic_stream_descriptor` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_stream_descriptor(io_service&)` constructor.

5.170.25 posix::basic_stream_descriptor::read_some

Read some data from the descriptor.

```
template<  
    typename MutableBufferSequence>  
std::size_t read_some(  
    const MutableBufferSequence & buffers);  
  
template<  
    typename MutableBufferSequence>  
std::size_t read_some(  
    const MutableBufferSequence & buffers,  
    asio::error_code & ec);
```

5.170.25.1 posix::basic_stream_descriptor::read_some (1 of 2 overloads)

Read some data from the descriptor.

```
template<  
    typename MutableBufferSequence>  
std::size_t read_some(  
    const MutableBufferSequence & buffers);
```

This function is used to read data from the stream descriptor. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be read.

Return Value

The number of bytes read.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The `read_some` operation may not read all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

Example

To read into a single data buffer use the `buffer` function as follows:

```
descriptor.read_some(asio::buffer(data, size));
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.170.25.2 posix::basic_stream_descriptor::read_some (2 of 2 overloads)

Read some data from the descriptor.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to read data from the stream descriptor. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be read.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes read. Returns 0 if an error occurred.

Remarks

The `read_some` operation may not read all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

5.170.26 posix::basic_stream_descriptor::release

Inherited from posix::basic_descriptor.

Release ownership of the native descriptor implementation.

```
native_handle_type release();
```

This function may be used to obtain the underlying representation of the descriptor. After calling this function, `is_open()` returns false. The caller is responsible for closing the descriptor.

All outstanding asynchronous read or write operations will finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

5.170.27 posix::basic_stream_descriptor::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`.) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.170.28 posix::basic_stream_descriptor::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef StreamDescriptorService service_type;
```

Requirements

Header: asioposix/basic_stream_descriptor.hpp

Convenience header: asio.hpp

5.170.29 posix::basic_stream_descriptor::write_some

Write some data to the descriptor.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.170.29.1 posix::basic_stream_descriptor::write_some (1 of 2 overloads)

Write some data to the descriptor.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

This function is used to write data to the stream descriptor. The function call will block until one or more bytes of the data has been written successfully, or until an error occurs.

Parameters

buffers One or more data buffers to be written to the descriptor.

Return Value

The number of bytes written.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The `write_some` operation may not transmit all of the data to the peer. Consider using the `write` function if you need to ensure that all data is written before the blocking operation completes.

Example

To write a single data buffer use the `buffer` function as follows:

```
descriptor.write_some(asio::buffer(data, size));
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.170.29.2 posix::basic_stream_descriptor::write_some (2 of 2 overloads)

Write some data to the descriptor.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to write data to the stream descriptor. The function call will block until one or more bytes of the data has been written successfully, or until an error occurs.

Parameters

buffers One or more data buffers to be written to the descriptor.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes written. Returns 0 if an error occurred.

Remarks

The `write_some` operation may not transmit all of the data to the peer. Consider using the `write` function if you need to ensure that all data is written before the blocking operation completes.

5.171 posix::descriptor_base

The `posix::descriptor_base` class is used as a base for the `posix::basic_stream_descriptor` class template so that we have a common place to define the associated IO control commands.

```
class descriptor_base
```

Types

Name	Description
<code>bytes_readable</code>	IO control command to get the amount of data that can be read without blocking.
<code>non_blocking_io</code>	(Deprecated: Use <code>non_blocking()</code> .) IO control command to set the blocking mode of the descriptor.

Protected Member Functions

Name	Description
<code>~descriptor_base</code>	Protected destructor to prevent deletion through this type.

Requirements

Header: `asio/posix/descriptor_base.hpp`

Convenience header: `asio.hpp`

5.171.1 posix::descriptor_base::bytes_readable

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::posix::stream_descriptor descriptor(io_service);
...
asio::descriptor_base::bytes_readable command(true);
descriptor.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: `asio/posix/descriptor_base.hpp`

Convenience header: `asio.hpp`

5.171.2 posix::descriptor_base::non_blocking_io

(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the descriptor.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::posix::stream_descriptor descriptor(io_service);
...
asio::descriptor_base::non_blocking_io command(true);
descriptor.io_control(command);
```

Requirements

Header: asio/posix/descriptor_base.hpp

Convenience header: asio.hpp

5.171.3 posix::descriptor_base::~descriptor_base

Protected destructor to prevent deletion through this type.

```
~descriptor_base();
```

5.172 posix::stream_descriptor

Typedef for the typical usage of a stream-oriented descriptor.

```
typedef basic_stream_descriptor stream_descriptor;
```

Types

Name	Description
bytes_readable	IO control command to get the amount of data that can be read without blocking.
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_descriptor is always the lowest layer.
native_handle_type	The native representation of a descriptor.
native_type	(Deprecated: Use native_handle_type.) The native representation of a descriptor.
non_blocking_io	(Deprecated: Use non_blocking().) IO control command to set the blocking mode of the descriptor.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
<code>assign</code>	Assign an existing native descriptor to the descriptor.
<code>async_read_some</code>	Start an asynchronous read.
<code>async_write_some</code>	Start an asynchronous write.
<code>basic_stream_descriptor</code>	Construct a <code>basic_stream_descriptor</code> without opening it. Construct a <code>basic_stream_descriptor</code> on an existing native descriptor. Move-construct a <code>basic_stream_descriptor</code> from another.
<code>cancel</code>	Cancel all asynchronous operations associated with the descriptor.
<code>close</code>	Close the descriptor.
<code>get_io_service</code>	Get the <code>io_service</code> associated with the object.
<code>io_control</code>	Perform an IO control command on the descriptor.
<code>is_open</code>	Determine whether the descriptor is open.
<code>lowest_layer</code>	Get a reference to the lowest layer. Get a const reference to the lowest layer.
<code>native</code>	(Deprecated: Use <code>native_handle()</code>) Get the native descriptor representation.
<code>native_handle</code>	Get the native descriptor representation.
<code>native_non_blocking</code>	Gets the non-blocking mode of the native descriptor implementation. Sets the non-blocking mode of the native descriptor implementation.
<code>non_blocking</code>	Gets the non-blocking mode of the descriptor. Sets the non-blocking mode of the descriptor.
<code>operator=</code>	Move-assign a <code>basic_stream_descriptor</code> from another.
<code>read_some</code>	Read some data from the descriptor.
<code>release</code>	Release ownership of the native descriptor implementation.
<code>write_some</code>	Write some data to the descriptor.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `posix::basic_stream_descriptor` class template provides asynchronous and blocking stream-oriented descriptor functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/posix/stream_descriptor.hpp`

Convenience header: `asio.hpp`

5.173 posix::stream_descriptor_service

Default service implementation for a stream descriptor.

```
class stream_descriptor_service :
    public io_service::service
```

Types

Name	Description
implementation_type	The type of a stream descriptor implementation.
native_handle_type	The native descriptor type.
native_type	(Deprecated: Use native_handle_type.) The native descriptor type.

Member Functions

Name	Description
assign	Assign an existing native descriptor to a stream descriptor.
async_read_some	Start an asynchronous read.
async_write_some	Start an asynchronous write.
cancel	Cancel all asynchronous operations associated with the descriptor.
close	Close a stream descriptor implementation.
construct	Construct a new stream descriptor implementation.
destroy	Destroy a stream descriptor implementation.
get_io_service	Get the io_service object that owns the service.
io_control	Perform an IO control command on the descriptor.
is_open	Determine whether the descriptor is open.
move_assign	Move-assign from another stream descriptor implementation.
move_construct	Move-construct a new stream descriptor implementation.
native	(Deprecated: Use native_handle().) Get the native descriptor implementation.
native_handle	Get the native descriptor implementation.
native_non_blocking	Gets the non-blocking mode of the native descriptor implementation. Sets the non-blocking mode of the native descriptor implementation.
non_blocking	Gets the non-blocking mode of the descriptor. Sets the non-blocking mode of the descriptor.
read_some	Read some data from the stream.
release	Release ownership of the native descriptor implementation.
stream_descriptor_service	Construct a new stream descriptor service for the specified io_service.
write_some	Write the given data to the stream.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio posix stream_descriptor_service.hpp

Convenience header: asio.hpp

5.173.1 posix::stream_descriptor_service::assign

Assign an existing native descriptor to a stream descriptor.

```
asio::error_code assign(
    implementation_type & impl,
    const native_handle_type & native_descriptor,
    asio::error_code & ec);
```

5.173.2 posix::stream_descriptor_service::async_read_some

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

5.173.3 posix::stream_descriptor_service::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

5.173.4 posix::stream_descriptor_service::cancel

Cancel all asynchronous operations associated with the descriptor.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.173.5 posix::stream_descriptor_service::close

Close a stream descriptor implementation.

```
asio::error_code close(
    implementation_type & impl,
    asio::error_code & ec);
```

5.173.6 posix::stream_descriptor_service::construct

Construct a new stream descriptor implementation.

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

5.173.7 posix::stream_descriptor_service::destroy

Destroy a stream descriptor implementation.

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

5.173.8 posix::stream_descriptor_service::get_io_service

Inherited from io_service.

Get the **io_service** object that owns the service.

```
asio::io_service & get_io_service();
```

5.173.9 posix::stream_descriptor_service::id

The unique service identifier.

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

5.173.10 posix::stream_descriptor_service::implementation_type

The type of a stream descriptor implementation.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asioposix/stream_descriptor_service.hpp

Convenience header: asio.hpp

5.173.11 posix::stream_descriptor_service::io_control

Perform an IO control command on the descriptor.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    implementation_type & impl,
    IoControlCommand & command,
    asio::error_code & ec);
```

5.173.12 posix::stream_descriptor_service::is_open

Determine whether the descriptor is open.

```
bool is_open(
    const implementation_type & impl) const;
```

5.173.13 posix::stream_descriptor_service::move_assign

Move-assign from another stream descriptor implementation.

```
void move_assign(
    implementation_type & impl,
    stream_descriptor_service & other_service,
    implementation_type & other_impl);
```

5.173.14 posix::stream_descriptor_service::move_construct

Move-construct a new stream descriptor implementation.

```
void move_construct(
    implementation_type & impl,
    implementation_type & other_impl);
```

5.173.15 posix::stream_descriptor_service::native

(Deprecated: Use native_handle().) Get the native descriptor implementation.

```
native_type native(
    implementation_type & impl);
```

5.173.16 posix::stream_descriptor_service::native_handle

Get the native descriptor implementation.

```
native_handle_type native_handle(
    implementation_type & impl);
```

5.173.17 posix::stream_descriptor_service::native_handle_type

The native descriptor type.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio posix stream_descriptor_service.hpp

Convenience header: asio.hpp

5.173.18 posix::stream_descriptor_service::native_non_blocking

Gets the non-blocking mode of the native descriptor implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the native descriptor implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.173.18.1 posix::stream_descriptor_service::native_non_blocking (1 of 2 overloads)

Gets the non-blocking mode of the native descriptor implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

5.173.18.2 posix::stream_descriptor_service::native_non_blocking (2 of 2 overloads)

Sets the non-blocking mode of the native descriptor implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.173.19 posix::stream_descriptor_service::native_type

(Deprecated: Use native_handle_type.) The native descriptor type.

```
typedef implementation_defined native_type;
```

Requirements

Header: asio posix stream_descriptor_service.hpp

Convenience header: asio.hpp

5.173.20 posix::stream_descriptor_service::non_blocking

Gets the non-blocking mode of the descriptor.

```
bool non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the descriptor.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.173.20.1 posix::stream_descriptor_service::non_blocking (1 of 2 overloads)

Gets the non-blocking mode of the descriptor.

```
bool non_blocking(
    const implementation_type & impl) const;
```

5.173.20.2 posix::stream_descriptor_service::non_blocking (2 of 2 overloads)

Sets the non-blocking mode of the descriptor.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.173.21 posix::stream_descriptor_service::read_some

Read some data from the stream.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.173.22 posix::stream_descriptor_service::release

Release ownership of the native descriptor implementation.

```
native_handle_type release(
    implementation_type & impl);
```

5.173.23 posix::stream_descriptor_service::stream_descriptor_service

Construct a new stream descriptor service for the specified [io_service](#).

```
stream_descriptor_service(
    asio::io_service & io_service);
```

5.173.24 posix::stream_descriptor_service::write_some

Write the given data to the stream.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.174 raw_socket_service

Default service implementation for a raw socket.

```
template<
    typename Protocol>
class raw_socket_service :
    public io_service::service
```

Types

Name	Description
endpoint_type	The endpoint type.
implementation_type	The type of a raw socket.
native_handle_type	The native socket type.
native_type	(Deprecated: Use native_handle_type.) The native socket type.
protocol_type	The protocol type.

Member Functions

Name	Description
assign	Assign an existing native socket to a raw socket.
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive.
async_receive_from	Start an asynchronous receive that will get the endpoint of the sender.
async_send	Start an asynchronous send.
async_send_to	Start an asynchronous send.

Name	Description
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.
bind	
cancel	Cancel all asynchronous operations associated with the socket.
close	Close a raw socket implementation.
connect	Connect the raw socket to the specified endpoint.
construct	Construct a new raw socket implementation.
converting_move_construct	Move-construct a new raw socket implementation from another protocol type.
destroy	Destroy a raw socket implementation.
get_io_service	Get the io_service object that owns the service.
get_option	Get a socket option.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint.
move_assign	Move-assign from another raw socket implementation.
move_construct	Move-construct a new raw socket implementation.
native	(Deprecated: Use native_handle().) Get the native socket implementation.
native_handle	Get the native socket implementation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	
raw_socket_service	Construct a new raw socket service for the specified io_service.
receive	Receive some data from the peer.
receive_from	Receive raw data with the endpoint of the sender.

Name	Description
remote_endpoint	Get the remote endpoint.
send	Send the given data to the peer.
send_to	Send raw data to the specified endpoint.
set_option	Set a socket option.
shutdown	Disable sends or receives on the socket.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/raw_socket_service.hpp

Convenience header: asio.hpp

5.174.1 raw_socket_service::assign

Assign an existing native socket to a raw socket.

```
asio::error_code assign(
    implementation_type & impl,
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.174.2 raw_socket_service::async_connect

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
void-or-deduced async_connect(
    implementation_type & impl,
    const endpoint_type & peer_endpoint,
    ConnectHandler handler);
```

5.174.3 raw_socket_service::async_receive

Start an asynchronous receive.

```

template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    ReadHandler handler);

```

5.174.4 raw_socket_service::async_receive_from

Start an asynchronous receive that will get the endpoint of the sender.

```

template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive_from(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    ReadHandler handler);

```

5.174.5 raw_socket_service::async_send

Start an asynchronous send.

```

template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);

```

5.174.6 raw_socket_service::async_send_to

Start an asynchronous send.

```

template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send_to(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    WriteHandler handler);

```

5.174.7 raw_socket_service::at_mark

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.174.8 raw_socket_service::available

Determine the number of bytes available for reading.

```
std::size_t available(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.174.9 raw_socket_service::bind

```
asio::error_code bind(
    implementation_type & impl,
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.174.10 raw_socket_service::cancel

Cancel all asynchronous operations associated with the socket.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.174.11 raw_socket_service::close

Close a raw socket implementation.

```
asio::error_code close(
    implementation_type & impl,
    asio::error_code & ec);
```

5.174.12 raw_socket_service::connect

Connect the raw socket to the specified endpoint.

```
asio::error_code connect(
    implementation_type & impl,
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.174.13 raw_socket_service::construct

Construct a new raw socket implementation.

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

5.174.14 raw_socket_service::converting_move_construct

Move-construct a new raw socket implementation from another protocol type.

```
template<
    typename Protocol1>
void converting_move_construct(
    implementation_type & impl,
    typename raw_socket_service< Protocol1 >::implementation_type & other_impl,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.174.15 raw_socket_service::destroy

Destroy a raw socket implementation.

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

5.174.16 raw_socket_service::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/raw_socket_service.hpp

Convenience header: asio.hpp

5.174.17 raw_socket_service::get_io_service

Inherited from io_service.

Get the **io_service** object that owns the service.

```
asio::io_service & get_io_service();
```

5.174.18 raw_socket_service::get_option

Get a socket option.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    const implementation_type & impl,
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.174.19 raw_socket_service::id

The unique service identifier.

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

5.174.20 raw_socket_service::implementation_type

The type of a raw socket.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asio/raw_socket_service.hpp

Convenience header: asio.hpp

5.174.21 raw_socket_service::io_control

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    implementation_type & impl,
    IoControlCommand & command,
    asio::error_code & ec);
```

5.174.22 raw_socket_service::is_open

Determine whether the socket is open.

```
bool is_open(
    const implementation_type & impl) const;
```

5.174.23 raw_socket_service::local_endpoint

Get the local endpoint.

```
endpoint_type local_endpoint(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.174.24 raw_socket_service::move_assign

Move-assign from another raw socket implementation.

```
void move_assign(
    implementation_type & impl,
    raw_socket_service & other_service,
    implementation_type & other_impl);
```

5.174.25 raw_socket_service::move_construct

Move-construct a new raw socket implementation.

```
void move_construct(
    implementation_type & impl,
    implementation_type & other_impl);
```

5.174.26 raw_socket_service::native

(Deprecated: Use native_handle().) Get the native socket implementation.

```
native_type native(
    implementation_type & impl);
```

5.174.27 raw_socket_service::native_handle

Get the native socket implementation.

```
native_handle_type native_handle(
    implementation_type & impl);
```

5.174.28 raw_socket_service::native_handle_type

The native socket type.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/raw_socket_service.hpp

Convenience header: asio.hpp

5.174.29 raw_socket_service::native_non_blocking

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the native socket implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.174.29.1 raw_socket_service::native_non_blocking (1 of 2 overloads)

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

5.174.29.2 raw_socket_service::native_non_blocking (2 of 2 overloads)

Sets the non-blocking mode of the native socket implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.174.30 raw_socket_service::native_type

(Deprecated: Use native_handle_type.) The native socket type.

```
typedef implementation_defined native_type;
```

Requirements

Header: asio/raw_socket_service.hpp

Convenience header: asio.hpp

5.174.31 raw_socket_service::non_blocking

Gets the non-blocking mode of the socket.

```
bool non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.174.31.1 raw_socket_service::non_blocking (1 of 2 overloads)

Gets the non-blocking mode of the socket.

```
bool non_blocking(
    const implementation_type & impl) const;
```

5.174.31.2 raw_socket_service::non_blocking (2 of 2 overloads)

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.174.32 raw_socket_service::open

```
asio::error_code open(
    implementation_type & impl,
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.174.33 raw_socket_service::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/raw_socket_service.hpp

Convenience header: asio.hpp

5.174.34 raw_socket_service::raw_socket_service

Construct a new raw socket service for the specified **io_service**.

```
raw_socket_service(
   asio::io_service & io_service);
```

5.174.35 raw_socket_service::receive

Receive some data from the peer.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.174.36 raw_socket_service::receive_from

Receive raw data with the endpoint of the sender.

```
template<
    typename MutableBufferSequence>
std::size_t receive_from(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.174.37 raw_socket_service::remote_endpoint

Get the remote endpoint.

```
endpoint_type remote_endpoint(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.174.38 raw_socket_service::send

Send the given data to the peer.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.174.39 raw_socket_service::send_to

Send raw data to the specified endpoint.

```
template<
    typename ConstBufferSequence>
std::size_t send_to(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.174.40 raw_socket_service::set_option

Set a socket option.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    implementation_type & impl,
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.174.41 raw_socket_service::shutdown

Disable sends or receives on the socket.

```
asio::error_code shutdown(
    implementation_type & impl,
    socket_base::shutdown_type what,
    asio::error_code & ec);
```

5.175 read

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename MutableBufferSequence>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers);

template<
    typename SyncReadStream,
    typename MutableBufferSequence>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers,
    asio::error_code & ec);

template<
    typename SyncReadStream,
    typename MutableBufferSequence,
```

```

    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition);

template<
    typename SyncReadStream,
    typename MutableBufferSequence,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec);

template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read(
    SyncReadStream & s,
    basic_streambuf<Allocator> & b);

template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read(
    SyncReadStream & s,
    basic_streambuf<Allocator> & b,
    asio::error_code & ec);

template<
    typename SyncReadStream,
    typename Allocator,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    basic_streambuf<Allocator> & b,
    CompletionCondition completion_condition);

template<
    typename SyncReadStream,
    typename Allocator,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    basic_streambuf<Allocator> & b,
    CompletionCondition completion_condition,
    asio::error_code & ec);

```

Requirements

Header: asio/read.hpp

Convenience header: asio.hpp

5.175.1 read (1 of 8 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename MutableBufferSequence>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers);
```

This function is used to read a certain number of bytes of data from a stream. The call will block until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function.

Parameters

s The stream from which the data is to be read. The type must support the `SyncReadStream` concept.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the stream.

Return Value

The number of bytes transferred.

Exceptions

`asio::system_error` Thrown on failure.

Example

To read into a single data buffer use the `buffer` function as follows:

```
asio::read(s, asio::buffer(data, size));
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

Remarks

This overload is equivalent to calling:

```
asio::read(
    s, buffers,
    asio::transfer_all());
```

5.175.2 `read` (2 of 8 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename MutableBufferSequence>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to read a certain number of bytes of data from a stream. The call will block until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function.

Parameters

s The stream from which the data is to be read. The type must support the `SyncReadStream` concept.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the stream.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes transferred.

Example

To read into a single data buffer use the `buffer` function as follows:

```
asio::read(s, asio::buffer(data, size), ec);
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

Remarks

This overload is equivalent to calling:

```
asio::read(
    s, buffers,
    asio::transfer_all(), ec);
```

5.175.3 `read` (3 of 8 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename MutableBufferSequence,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition);
```

This function is used to read a certain number of bytes of data from a stream. The call will block until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function.

Parameters

s The stream from which the data is to be read. The type must support the `SyncReadStream` concept.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the stream.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest read_some operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the stream's `read_some` function.

Return Value

The number of bytes transferred.

Exceptions

`asio::system_error` Thrown on failure.

Example

To read into a single data buffer use the `buffer` function as follows:

```
asio::read(s, asio::buffer(data, size),
    asio::transfer_at_least(32));
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.175.4 read (4 of 8 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename MutableBufferSequence,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec);
```

This function is used to read a certain number of bytes of data from a stream. The call will block until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The completion_condition function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's read_some function.

Parameters

s The stream from which the data is to be read. The type must support the SyncReadStream concept.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the stream.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest read_some operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the stream's read_some function.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes read. If an error occurs, returns the total number of bytes successfully transferred prior to the error.

5.175.5 read (5 of 8 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read(
    SyncReadStream & s,
    basic_streambuf<Allocator> & b);
```

This function is used to read a certain number of bytes of data from a stream. The call will block until one of the following conditions is true:

- The supplied buffer is full (that is, it has reached maximum size).
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function.

Parameters

s The stream from which the data is to be read. The type must support the `SyncReadStream` concept.

b The `basic_streambuf` object into which the data will be read.

Return Value

The number of bytes transferred.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

This overload is equivalent to calling:

```
asio::read(
    s, b,
    asio::transfer_all());
```

5.175.6 `read` (6 of 8 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read(
    SyncReadStream & s,
    basic_streambuf< Allocator > & b,
    asio::error_code & ec);
```

This function is used to read a certain number of bytes of data from a stream. The call will block until one of the following conditions is true:

- The supplied buffer is full (that is, it has reached maximum size).
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function.

Parameters

- s** The stream from which the data is to be read. The type must support the SyncReadStream concept.
- b** The `basic_streambuf` object into which the data will be read.
- ec** Set to indicate what error occurred, if any.

Return Value

The number of bytes transferred.

Remarks

This overload is equivalent to calling:

```
asio::read(  
    s, b,  
    asio::transfer_all(), ec);
```

5.175.7 `read` (7 of 8 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<  
    typename SyncReadStream,  
    typename Allocator,  
    typename CompletionCondition>  
std::size_t read(  
    SyncReadStream & s,  
    basic_streambuf< Allocator > & b,  
    CompletionCondition completion_condition);
```

This function is used to read a certain number of bytes of data from a stream. The call will block until one of the following conditions is true:

- The supplied buffer is full (that is, it has reached maximum size).
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function.

Parameters

- s** The stream from which the data is to be read. The type must support the SyncReadStream concept.
- b** The `basic_streambuf` object into which the data will be read.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(  
    // Result of latest read_some operation.  
    const asio::error_code& error,  
  
    // Number of bytes transferred so far.  
    std::size_t bytes_transferred  
) ;
```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the stream's `read_some` function.

Return Value

The number of bytes transferred.

Exceptions

asio::system_error Thrown on failure.

5.175.8 read (8 of 8 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename Allocator,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    asio::error_code & ec);
```

This function is used to read a certain number of bytes of data from a stream. The call will block until one of the following conditions is true:

- The supplied buffer is full (that is, it has reached maximum size).
- The completion_condition function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's read_some function.

Parameters

s The stream from which the data is to be read. The type must support the SyncReadStream concept.

b The **basic_streambuf** object into which the data will be read.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest read_some operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the stream's read_some function.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes read. If an error occurs, returns the total number of bytes successfully transferred prior to the error.

5.176 `read_at`

Attempt to read a certain amount of data at the specified offset before returning.

```
template<
    typename SyncRandomAccessReadDevice,
    typename MutableBufferSequence>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers);
```



```
template<
    typename SyncRandomAccessReadDevice,
    typename MutableBufferSequence>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```



```
template<
    typename SyncRandomAccessReadDevice,
    typename MutableBufferSequence,
    typename CompletionCondition>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition);
```



```
template<
    typename SyncRandomAccessReadDevice,
    typename MutableBufferSequence,
    typename CompletionCondition>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec);
```



```
template<
    typename SyncRandomAccessReadDevice,
    typename Allocator>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b);
```



```
template<
    typename SyncRandomAccessReadDevice,
    typename Allocator>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
```

```

asio::error_code & ec);

template<
    typename SyncRandomAccessReadDevice,
    typename Allocator,
    typename CompletionCondition>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition);

template<
    typename SyncRandomAccessReadDevice,
    typename Allocator,
    typename CompletionCondition>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    asio::error_code & ec);

```

Requirements

Header: asio/read_at.hpp

Convenience header: asio.hpp

5.176.1 read_at (1 of 8 overloads)

Attempt to read a certain amount of data at the specified offset before returning.

```

template<
    typename SyncRandomAccessReadDevice,
    typename MutableBufferSequence>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers);

```

This function is used to read a certain number of bytes of data from a random access device at the specified offset. The call will block until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the device's `read_some_at` function.

Parameters

d The device from which the data is to be read. The type must support the `SyncRandomAccessReadDevice` concept.

offset The offset at which the data will be read.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the device.

Return Value

The number of bytes transferred.

Exceptions

asio::system_error Thrown on failure.

Example

To read into a single data buffer use the **buffer** function as follows:

```
asio::read_at(d, 42, asio::buffer(data, size));
```

See the **buffer** documentation for information on reading into multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

Remarks

This overload is equivalent to calling:

```
asio::read_at(
    d, 42, buffers,
    asio::transfer_all());
```

5.176.2 **read_at (2 of 8 overloads)**

Attempt to read a certain amount of data at the specified offset before returning.

```
template<
    typename SyncRandomAccessReadDevice,
    typename MutableBufferSequence>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to read a certain number of bytes of data from a random access device at the specified offset. The call will block until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the device's `read_some_at` function.

Parameters

d The device from which the data is to be read. The type must support the `SyncRandomAccessReadDevice` concept.

offset The offset at which the data will be read.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the device.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes transferred.

Example

To read into a single data buffer use the **buffer** function as follows:

```
asio::read_at(d, 42,
    asio::buffer(data, size), ec);
```

See the **buffer** documentation for information on reading into multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

Remarks

This overload is equivalent to calling:

```
asio::read_at(
    d, 42, buffers,
    asio::transfer_all(), ec);
```

5.176.3 **read_at (3 of 8 overloads)**

Attempt to read a certain amount of data at the specified offset before returning.

```
template<
    typename SyncRandomAccessReadDevice,
    typename MutableBufferSequence,
    typename CompletionCondition>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition);
```

This function is used to read a certain number of bytes of data from a random access device at the specified offset. The call will block until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the device's `read_some_at` function.

Parameters

d The device from which the data is to be read. The type must support the `SyncRandomAccessReadDevice` concept.

offset The offset at which the data will be read.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the device.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```

std::size_t completion_condition(
    // Result of latest read_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);

```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the device's `read_some_at` function.

Return Value

The number of bytes transferred.

Exceptions

`asio::system_error` Thrown on failure.

Example

To read into a single data buffer use the `buffer` function as follows:

```

asio::read_at(d, 42, asio::buffer(data, size),
    asio::transfer_at_least(32));

```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.176.4 `read_at` (4 of 8 overloads)

Attempt to read a certain amount of data at the specified offset before returning.

```

template<
    typename SyncRandomAccessReadDevice,
    typename MutableBufferSequence,
    typename CompletionCondition>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec);

```

This function is used to read a certain number of bytes of data from a random access device at the specified offset. The call will block until one of the following conditions is true:

- The supplied buffers are full. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the device's `read_some_at` function.

Parameters

d The device from which the data is to be read. The type must support the SyncRandomAccessReadDevice concept.

offset The offset at which the data will be read.

buffers One or more buffers into which the data will be read. The sum of the buffer sizes indicates the maximum number of bytes to read from the device.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest read_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the device's read_some_at function.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes read. If an error occurs, returns the total number of bytes successfully transferred prior to the error.

5.176.5 `read_at` (5 of 8 overloads)

Attempt to read a certain amount of data at the specified offset before returning.

```
template<
    typename SyncRandomAccessReadDevice,
    typename Allocator>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf<Allocator> & b);
```

This function is used to read a certain number of bytes of data from a random access device at the specified offset. The call will block until one of the following conditions is true:

- An error occurred.

This operation is implemented in terms of zero or more calls to the device's read_some_at function.

Parameters

d The device from which the data is to be read. The type must support the SyncRandomAccessReadDevice concept.

offset The offset at which the data will be read.

b The `basic_streambuf` object into which the data will be read.

Return Value

The number of bytes transferred.

Exceptions

asio::system_error Thrown on failure.

Remarks

This overload is equivalent to calling:

```
asio::read_at(
    d, 42, b,
    asio::transfer_all());
```

5.176.6 `read_at` (6 of 8 overloads)

Attempt to read a certain amount of data at the specified offset before returning.

```
template<
    typename SyncRandomAccessReadDevice,
    typename Allocator>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf<Allocator> & b,
    asio::error_code & ec);
```

This function is used to read a certain number of bytes of data from a random access device at the specified offset. The call will block until one of the following conditions is true:

- An error occurred.

This operation is implemented in terms of zero or more calls to the device's `read_some_at` function.

Parameters

d The device from which the data is to be read. The type must support the `SyncRandomAccessReadDevice` concept.

offset The offset at which the data will be read.

b The `basic_streambuf` object into which the data will be read.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes transferred.

Remarks

This overload is equivalent to calling:

```
asio::read_at(
    d, 42, b,
    asio::transfer_all(), ec);
```

5.176.7 `read_at` (7 of 8 overloads)

Attempt to read a certain amount of data at the specified offset before returning.

```
template<
    typename SyncRandomAccessReadDevice,
    typename Allocator,
    typename CompletionCondition>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition);
```

This function is used to read a certain number of bytes of data from a random access device at the specified offset. The call will block until one of the following conditions is true:

- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the device's `read_some_at` function.

Parameters

d The device from which the data is to be read. The type must support the `SyncRandomAccessReadDevice` concept.

offset The offset at which the data will be read.

b The `basic_streambuf` object into which the data will be read.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest read_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the device's `read_some_at` function.

Return Value

The number of bytes transferred.

Exceptions

asio::system_error Thrown on failure.

5.176.8 `read_at` (8 of 8 overloads)

Attempt to read a certain amount of data at the specified offset before returning.

```
template<
    typename SyncRandomAccessReadDevice,
    typename Allocator,
    typename CompletionCondition>
std::size_t read_at(
    SyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    asio::error_code & ec);
```

This function is used to read a certain number of bytes of data from a random access device at the specified offset. The call will block until one of the following conditions is true:

- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the device's `read_some_at` function.

Parameters

d The device from which the data is to be read. The type must support the `SyncRandomAccessReadDevice` concept.

offset The offset at which the data will be read.

b The `basic_streambuf` object into which the data will be read.

completion_condition The function object to be called to determine whether the read operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest read_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the read operation is complete. A non-zero return value indicates the maximum number of bytes to be read on the next call to the device's `read_some_at` function.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes read. If an error occurs, returns the total number of bytes successfully transferred prior to the error.

5.177 `read_until`

Read data into a `streambuf` until it contains a delimiter, matches a regular expression, or a function object indicates a match.

```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
   asio::basic_streambuf< Allocator > & b,
    char delim);
```



```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    char delim,
    asio::error_code & ec);
```



```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    const std::string & delim);
```



```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    const std::string & delim,
    asio::error_code & ec);
```



```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    const boost::regex & expr);
```



```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    const boost::regex & expr,
    asio::error_code & ec);
```



```
template<
```

```

typename SyncReadStream,
typename Allocator,
typename MatchCondition>
std::size_t read_until(
    SyncReadStream & s,
   asio::basic_streambuf< Allocator > & b,
    MatchCondition match_condition,
    typename enable_if< is_match_condition< MatchCondition >::value >::type * = 0);

template<
    typename SyncReadStream,
    typename Allocator,
    typename MatchCondition>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    MatchCondition match_condition,
    asio::error_code & ec,
    typename enable_if< is_match_condition< MatchCondition >::value >::type * = 0);

```

Requirements

Header: asio/read_until.hpp

Convenience header: asio.hpp

5.177.1 `read_until` (1 of 8 overloads)

Read data into a streambuf until it contains a specified delimiter.

```

template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    char delim);

```

This function is used to read data into the specified streambuf until the streambuf's get area contains the specified delimiter. The call will block until one of the following conditions is true:

- The get area of the streambuf contains the specified delimiter.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function. If the streambuf's get area already contains the delimiter, the function returns immediately.

Parameters

s The stream from which the data is to be read. The type must support the `SyncReadStream` concept.

b A streambuf object into which the data will be read.

delim The delimiter character.

Return Value

The number of bytes in the streambuf's get area up to and including the delimiter.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

After a successful `read_until` operation, the streambuf may contain additional data beyond the delimiter. An application will typically leave that data in the streambuf for a subsequent `read_until` operation to examine.

Example

To read data into a streambuf until a newline is encountered:

```
asio::streambuf b;
asio::read_until(s, b, '\n');
std::istream is(&b);
std::string line;
std::getline(is, line);
```

After the `read_until` operation completes successfully, the buffer `b` contains the delimiter:

```
{ 'a', 'b', ..., 'c', '\n', 'd', 'e', ... }
```

The call to `std::getline` then extracts the data up to and including the delimiter, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\n' }
```

The remaining data is left in the buffer `b` as follows:

```
{ 'd', 'e', ... }
```

This data may be the start of a new line, to be extracted by a subsequent `read_until` operation.

5.177.2 `read_until` (2 of 8 overloads)

Read data into a streambuf until it contains a specified delimiter.

```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf<Allocator> & b,
    char delim,
    asio::error_code & ec);
```

This function is used to read data into the specified streambuf until the streambuf's get area contains the specified delimiter. The call will block until one of the following conditions is true:

- The get area of the streambuf contains the specified delimiter.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function. If the streambuf's get area already contains the delimiter, the function returns immediately.

Parameters

- s** The stream from which the data is to be read. The type must support the SyncReadStream concept.
- b** A streambuf object into which the data will be read.
- delim** The delimiter character.
- ec** Set to indicate what error occurred, if any.

Return Value

The number of bytes in the streambuf's get area up to and including the delimiter. Returns 0 if an error occurred.

Remarks

After a successful read_until operation, the streambuf may contain additional data beyond the delimiter. An application will typically leave that data in the streambuf for a subsequent read_until operation to examine.

5.177.3 `read_until` (3 of 8 overloads)

Read data into a streambuf until it contains a specified delimiter.

```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
   asio::basic_streambuf<Allocator> & b,
    const std::string & delim);
```

This function is used to read data into the specified streambuf until the streambuf's get area contains the specified delimiter. The call will block until one of the following conditions is true:

- The get area of the streambuf contains the specified delimiter.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's read_some function. If the streambuf's get area already contains the delimiter, the function returns immediately.

Parameters

- s** The stream from which the data is to be read. The type must support the SyncReadStream concept.
- b** A streambuf object into which the data will be read.
- delim** The delimiter string.

Return Value

The number of bytes in the streambuf's get area up to and including the delimiter.

Exceptions

asio::system_error Thrown on failure.

Remarks

After a successful `read_until` operation, the `streambuf` may contain additional data beyond the delimiter. An application will typically leave that data in the `streambuf` for a subsequent `read_until` operation to examine.

Example

To read data into a `streambuf` until a newline is encountered:

```
asio::streambuf b;
asio::read_until(s, b, "\r\n");
std::istream is(&b);
std::string line;
std::getline(is, line);
```

After the `read_until` operation completes successfully, the buffer `b` contains the delimiter:

```
{ 'a', 'b', ..., 'c', '\r', '\n', 'd', 'e', ... }
```

The call to `std::getline` then extracts the data up to and including the delimiter, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r', '\n' }
```

The remaining data is left in the buffer `b` as follows:

```
{ 'd', 'e', ... }
```

This data may be the start of a new line, to be extracted by a subsequent `read_until` operation.

5.177.4 `read_until` (4 of 8 overloads)

Read data into a `streambuf` until it contains a specified delimiter.

```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    const std::string & delim,
    asio::error_code & ec);
```

This function is used to read data into the specified `streambuf` until the `streambuf`'s get area contains the specified delimiter. The call will block until one of the following conditions is true:

- The get area of the `streambuf` contains the specified delimiter.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function. If the `streambuf`'s get area already contains the delimiter, the function returns immediately.

Parameters

- s** The stream from which the data is to be read. The type must support the SyncReadStream concept.
- b** A streambuf object into which the data will be read.
- delim** The delimiter string.
- ec** Set to indicate what error occurred, if any.

Return Value

The number of bytes in the streambuf's get area up to and including the delimiter. Returns 0 if an error occurred.

Remarks

After a successful read_until operation, the streambuf may contain additional data beyond the delimiter. An application will typically leave that data in the streambuf for a subsequent read_until operation to examine.

5.177.5 `read_until` (5 of 8 overloads)

Read data into a streambuf until some part of the data it contains matches a regular expression.

```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf<Allocator> & b,
    const boost::regex & expr);
```

This function is used to read data into the specified streambuf until the streambuf's get area contains some data that matches a regular expression. The call will block until one of the following conditions is true:

- A substring of the streambuf's get area matches the regular expression.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's read_some function. If the streambuf's get area already contains data that matches the regular expression, the function returns immediately.

Parameters

- s** The stream from which the data is to be read. The type must support the SyncReadStream concept.
- b** A streambuf object into which the data will be read.
- expr** The regular expression.

Return Value

The number of bytes in the streambuf's get area up to and including the substring that matches the regular expression.

Exceptions

asio::system_error Thrown on failure.

Remarks

After a successful `read_until` operation, the `streambuf` may contain additional data beyond that which matched the regular expression. An application will typically leave that data in the `streambuf` for a subsequent `read_until` operation to examine.

Example

To read data into a `streambuf` until a CR-LF sequence is encountered:

```
asio::streambuf b;
asio::read_until(s, b, boost::regex("\r\n"));
std::istream is(&b);
std::string line;
std::getline(is, line);
```

After the `read_until` operation completes successfully, the buffer `b` contains the data which matched the regular expression:

```
{ 'a', 'b', ..., 'c', '\r', '\n', 'd', 'e', ... }
```

The call to `std::getline` then extracts the data up to and including the match, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r', '\n' }
```

The remaining data is left in the buffer `b` as follows:

```
{ 'd', 'e', ... }
```

This data may be the start of a new line, to be extracted by a subsequent `read_until` operation.

5.177.6 `read_until` (6 of 8 overloads)

Read data into a `streambuf` until some part of the data it contains matches a regular expression.

```
template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    const boost::regex & expr,
    asio::error_code & ec);
```

This function is used to read data into the specified `streambuf` until the `streambuf`'s get area contains some data that matches a regular expression. The call will block until one of the following conditions is true:

- A substring of the `streambuf`'s get area matches the regular expression.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function. If the `streambuf`'s get area already contains data that matches the regular expression, the function returns immediately.

Parameters

- s** The stream from which the data is to be read. The type must support the SyncReadStream concept.
- b** A streambuf object into which the data will be read.
- expr** The regular expression.
- ec** Set to indicate what error occurred, if any.

Return Value

The number of bytes in the streambuf's get area up to and including the substring that matches the regular expression. Returns 0 if an error occurred.

Remarks

After a successful read_until operation, the streambuf may contain additional data beyond that which matched the regular expression. An application will typically leave that data in the streambuf for a subsequent read_until operation to examine.

5.177.7 read_until (7 of 8 overloads)

Read data into a streambuf until a function object indicates a match.

```
template<
    typename SyncReadStream,
    typename Allocator,
    typename MatchCondition>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    MatchCondition match_condition,
    typename enable_if< is_match_condition< MatchCondition >::value >::type * = 0);
```

This function is used to read data into the specified streambuf until a user-defined match condition function object, when applied to the data contained in the streambuf, indicates a successful match. The call will block until one of the following conditions is true:

- The match condition function object returns a std::pair where the second element evaluates to true.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's read_some function. If the match condition function object already indicates a match, the function returns immediately.

Parameters

- s** The stream from which the data is to be read. The type must support the SyncReadStream concept.
- b** A streambuf object into which the data will be read.

match_condition The function object to be called to determine whether a match exists. The signature of the function object must be:

```
pair<iterator, bool> match_condition(iterator begin, iterator end);
```

where `iterator` represents the type:

```
buffers_iterator<basic_streambuf<Allocator>::const_buffers_type>
```

The iterator parameters `begin` and `end` define the range of bytes to be scanned to determine whether there is a match. The first member of the return value is an iterator marking one-past-the-end of the bytes that have been consumed by the match function. This iterator is used to calculate the `begin` parameter for any subsequent invocation of the match condition. The second member of the return value is true if a match has been found, false otherwise.

Return Value

The number of bytes in the streambuf's get area that have been fully consumed by the match function.

Exceptions

asio::system_error Thrown on failure.

Remarks

After a successful `read_until` operation, the streambuf may contain additional data beyond that which matched the function object. An application will typically leave that data in the streambuf for a subsequent

The default implementation of the `is_match_condition` type trait evaluates to true for function pointers and function objects with a `result_type` typedef. It must be specialised for other user-defined function objects.

Examples

To read data into a streambuf until whitespace is encountered:

```
typedef asio::buffers_iterator<  
    asio::streambuf::const_buffers_type> iterator;  
  
std::pair<iterator, bool>  
match_whitespace(iterator begin, iterator end)  
{  
    iterator i = begin;  
    while (i != end)  
        if (std::isspace(*i++))  
            return std::make_pair(i, true);  
    return std::make_pair(i, false);  
}  
...  
asio::streambuf b;  
asio::read_until(s, b, match_whitespace);
```

To read data into a streambuf until a matching character is found:

```
class match_char  
{  
public:  
    explicit match_char(char c) : c_(c) {}  
  
    template <typename Iterator>  
    std::pair<Iterator, bool> operator()(  
        Iterator begin, Iterator end) const  
{  
    Iterator i = begin;  
    while (i != end)  
        if (c_ == *i++)
```

```

        return std::make_pair(i, true);
        return std::make_pair(i, false);
    }

private:
    char c_;
};

namespaceasio{
    template<> struct is_match_condition<match_char>
        : public boost::true_type {};
} // namespaceasio
...
asio::streambuf b;
asio::read_until(s, b, match_char('a'));

```

5.177.8 `read_until` (8 of 8 overloads)

Read data into a streambuf until a function object indicates a match.

```

template<
    typename SyncReadStream,
    typename Allocator,
    typename MatchCondition>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf<Allocator> & b,
    MatchCondition match_condition,
    asio::error_code & ec,
    typename enable_if< is_match_condition<MatchCondition>::value >::type * = 0);

```

This function is used to read data into the specified streambuf until a user-defined match condition function object, when applied to the data contained in the streambuf, indicates a successful match. The call will block until one of the following conditions is true:

- The match condition function object returns a `std::pair` where the second element evaluates to true.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `read_some` function. If the match condition function object already indicates a match, the function returns immediately.

Parameters

s The stream from which the data is to be read. The type must support the `SyncReadStream` concept.

b A streambuf object into which the data will be read.

match_condition The function object to be called to determine whether a match exists. The signature of the function object must be:

```
pair<iterator, bool> match_condition(iterator begin, iterator end);
```

where `iterator` represents the type:

```
buffers_iterator<basic_streambuf<Allocator>::const_buffers_type>
```

The iterator parameters `begin` and `end` define the range of bytes to be scanned to determine whether there is a match. The first member of the return value is an iterator marking one-past-the-end of the bytes that have been consumed by the match function. This iterator is used to calculate the `begin` parameter for any subsequent invocation of the match condition. The second member of the return value is true if a match has been found, false otherwise.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes in the streambuf's get area that have been fully consumed by the match function. Returns 0 if an error occurred.

Remarks

After a successful read_until operation, the streambuf may contain additional data beyond that which matched the function object. An application will typically leave that data in the streambuf for a subsequent

The default implementation of the `is_match_condition` type trait evaluates to true for function pointers and function objects with a `result_type` typedef. It must be specialised for other user-defined function objects.

5.178 seq_packet_socket_service

Default service implementation for a sequenced packet socket.

```
template<
    typename Protocol>
class seq_packet_socket_service :
    public io_service::service
```

Types

Name	Description
<code>endpoint_type</code>	The endpoint type.
<code>implementation_type</code>	The type of a sequenced packet socket implementation.
<code>native_handle_type</code>	The native socket type.
<code>native_type</code>	(Deprecated: Use <code>native_handle_type</code> .) The native socket type.
<code>protocol_type</code>	The protocol type.

Member Functions

Name	Description
<code>assign</code>	Assign an existing native socket to a sequenced packet socket.
<code>async_connect</code>	Start an asynchronous connect.
<code>async_receive</code>	Start an asynchronous receive.
<code>async_send</code>	Start an asynchronous send.
<code>at_mark</code>	Determine whether the socket is at the out-of-band data mark.
<code>available</code>	Determine the number of bytes available for reading.

Name	Description
bind	Bind the sequenced packet socket to the specified local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close a sequenced packet socket implementation.
connect	Connect the sequenced packet socket to the specified endpoint.
construct	Construct a new sequenced packet socket implementation.
converting_move_construct	Move-construct a new sequenced packet socket implementation from another protocol type.
destroy	Destroy a sequenced packet socket implementation.
get_io_service	Get the io_service object that owns the service.
get_option	Get a socket option.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint.
move_assign	Move-assign from another sequenced packet socket implementation.
move_construct	Move-construct a new sequenced packet socket implementation.
native	(Deprecated: Use native_handle().) Get the native socket implementation.
native_handle	Get the native socket implementation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open a sequenced packet socket.
receive	Receive some data from the peer.
remote_endpoint	Get the remote endpoint.
send	Send the given data to the peer.

Name	Description
seq_packet_socket_service	Construct a new sequenced packet socket service for the specified io_service.
set_option	Set a socket option.
shutdown	Disable sends or receives on the socket.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/seq_packet_socket_service.hpp

Convenience header: asio.hpp

5.178.1 seq_packet_socket_service::assign

Assign an existing native socket to a sequenced packet socket.

```
asio::error_code assign(
    implementation_type & impl,
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.178.2 seq_packet_socket_service::async_connect

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
void-or-deduced async_connect(
    implementation_type & impl,
    const endpoint_type & peer_endpoint,
    ConnectHandler handler);
```

5.178.3 seq_packet_socket_service::async_receive

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
```

```
socket_base::message_flags in_flags,
socket_base::message_flags & out_flags,
ReadHandler handler);
```

5.178.4 seq_packet_socket_service::async_send

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);
```

5.178.5 seq_packet_socket_service::at_mark

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.178.6 seq_packet_socket_service::available

Determine the number of bytes available for reading.

```
std::size_t available(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.178.7 seq_packet_socket_service::bind

Bind the sequenced packet socket to the specified local endpoint.

```
asio::error_code bind(
    implementation_type & impl,
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.178.8 seq_packet_socket_service::cancel

Cancel all asynchronous operations associated with the socket.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.178.9 seq_packet_socket_service::close

Close a sequenced packet socket implementation.

```
asio::error_code close(
    implementation_type & impl,
    asio::error_code & ec);
```

5.178.10 seq_packet_socket_service::connect

Connect the sequenced packet socket to the specified endpoint.

```
asio::error_code connect(
    implementation_type & impl,
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.178.11 seq_packet_socket_service::construct

Construct a new sequenced packet socket implementation.

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

5.178.12 seq_packet_socket_service::converting_move_construct

Move-construct a new sequenced packet socket implementation from another protocol type.

```
template<
    typename Protocol1>
void converting_move_construct(
    implementation_type & impl,
    typename seq_packet_socket_service< Protocol1 >::implementation_type & other_impl,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.178.13 seq_packet_socket_service::destroy

Destroy a sequenced packet socket implementation.

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

5.178.14 seq_packet_socket_service::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/seq_packet_socket_service.hpp

Convenience header: asio.hpp

5.178.15 seq_packet_socket_service::get_io_service

Inherited from io_service.

Get the **io_service** object that owns the service.

```
asio::io_service & get_io_service();
```

5.178.16 seq_packet_socket_service::get_option

Get a socket option.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    const implementation_type & impl,
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.178.17 seq_packet_socket_service::id

The unique service identifier.

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

5.178.18 seq_packet_socket_service::implementation_type

The type of a sequenced packet socket implementation.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asio/seq_packet_socket_service.hpp

Convenience header: asio.hpp

5.178.19 seq_packet_socket_service::io_control

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    implementation_type & impl,
    IoControlCommand & command,
    asio::error_code & ec);
```

5.178.20 seq_packet_socket_service::is_open

Determine whether the socket is open.

```
bool is_open(
    const implementation_type & impl) const;
```

5.178.21 seq_packet_socket_service::local_endpoint

Get the local endpoint.

```
endpoint_type local_endpoint(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.178.22 seq_packet_socket_service::move_assign

Move-assign from another sequenced packet socket implementation.

```
void move_assign(
    implementation_type & impl,
    seq_packet_socket_service & other_service,
    implementation_type & other_impl);
```

5.178.23 seq_packet_socket_service::move_construct

Move-construct a new sequenced packet socket implementation.

```
void move_construct(
    implementation_type & impl,
    implementation_type & other_impl);
```

5.178.24 seq_packet_socket_service::native

(Deprecated: Use native_handle().) Get the native socket implementation.

```
native_type native(
    implementation_type & impl);
```

5.178.25 seq_packet_socket_service::native_handle

Get the native socket implementation.

```
native_handle_type native_handle(
    implementation_type & impl);
```

5.178.26 seq_packet_socket_service::native_handle_type

The native socket type.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/seq_packet_socket_service.hpp

Convenience header: asio.hpp

5.178.27 seq_packet_socket_service::native_non_blocking

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the native socket implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.178.27.1 seq_packet_socket_service::native_non_blocking (1 of 2 overloads)

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

5.178.27.2 seq_packet_socket_service::native_non_blocking (2 of 2 overloads)

Sets the non-blocking mode of the native socket implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.178.28 seq_packet_socket_service::native_type

(Deprecated: Use native_handle_type.) The native socket type.

```
typedef implementation_defined native_type;
```

Requirements

Header: asio/seq_packet_socket_service.hpp

Convenience header: asio.hpp

5.178.29 seq_packet_socket_service::non_blocking

Gets the non-blocking mode of the socket.

```
bool non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.178.29.1 seq_packet_socket_service::non_blocking (1 of 2 overloads)

Gets the non-blocking mode of the socket.

```
bool non_blocking(
    const implementation_type & impl) const;
```

5.178.29.2 seq_packet_socket_service::non_blocking (2 of 2 overloads)

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.178.30 seq_packet_socket_service::open

Open a sequenced packet socket.

```
asio::error_code open(
    implementation_type & impl,
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.178.31 seq_packet_socket_service::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/seq_packet_socket_service.hpp

Convenience header: asio.hpp

5.178.32 seq_packet_socket_service::receive

Receive some data from the peer.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    socket_base::message_flags in_flags,
    socket_base::message_flags & out_flags,
    asio::error_code & ec);
```

5.178.33 seq_packet_socket_service::remote_endpoint

Get the remote endpoint.

```
endpoint_type remote_endpoint(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.178.34 seq_packet_socket_service::send

Send the given data to the peer.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.178.35 seq_packet_socket_service::seq_packet_socket_service

Construct a new sequenced packet socket service for the specified [io_service](#).

```
seq_packet_socket_service(
    asio::io_service & io_service);
```

5.178.36 seq_packet_socket_service::set_option

Set a socket option.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    implementation_type & impl,
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.178.37 seq_packet_socket_service::shutdown

Disable sends or receives on the socket.

```
asio::error_code shutdown(
    implementation_type & impl,
    socket_base::shutdown_type what,
    asio::error_code & ec);
```

5.179 serial_port

Typedef for the typical usage of a serial port.

```
typedef basic_serial_port serial_port;
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A <code>basic_serial_port</code> is always the lowest layer.

Name	Description
native_handle_type	The native representation of a serial port.
native_type	(Deprecated: Use native_handle_type.) The native representation of a serial port.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native serial port to the serial port.
async_read_some	Start an asynchronous read.
async_write_some	Start an asynchronous write.
basic_serial_port	Construct a basic_serial_port without opening it. Construct and open a basic_serial_port. Construct a basic_serial_port on an existing native serial port. Move-construct a basic_serial_port from another.
cancel	Cancel all asynchronous operations associated with the serial port.
close	Close the serial port.
get_io_service	Get the io_service associated with the object.
get_option	Get an option from the serial port.
is_open	Determine whether the serial port is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native serial port representation.
native_handle	Get the native serial port representation.
open	Open the serial port using the specified device name.
operator=	Move-assign a basic_serial_port from another.
read_some	Read some data from the serial port.
send_break	Send a break sequence to the serial port.
set_option	Set an option on the serial port.

Name	Description
write_some	Write some data to the serial port.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_serial_port` class template provides functionality that is common to all serial ports.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/serial_port.hpp`

Convenience header: `asio.hpp`

5.180 serial_port_base

The `serial_port_base` class is used as a base for the `basic_serial_port` class template so that we have a common place to define the serial port options.

```
class serial_port_base
```

Types

Name	Description
baud_rate	Serial port option to permit changing the baud rate.

Name	Description
character_size	Serial port option to permit changing the character size.
flow_control	Serial port option to permit changing the flow control.
parity	Serial port option to permit changing the parity.
stop_bits	Serial port option to permit changing the number of stop bits.

Protected Member Functions

Name	Description
~serial_port_base	Protected destructor to prevent deletion through this type.

Requirements

Header: asio/serial_port_base.hpp

Convenience header: asio.hpp

5.180.1 serial_port_base::~serial_port_base

Protected destructor to prevent deletion through this type.

```
~serial_port_base();
```

5.181 serial_port_base::baud_rate

Serial port option to permit changing the baud rate.

```
class baud_rate
```

Member Functions

Name	Description
baud_rate	
load	
store	
value	

Implements changing the baud rate for a given serial port.

Requirements

Header: asio/serial_port_base.hpp

Convenience header: asio.hpp

5.181.1 serial_port_base::baud_rate::baud_rate

```
baud_rate(  
    unsigned int rate = 0);
```

5.181.2 serial_port_base::baud_rate::load

```
asio::error_code load(  
    const ASIO_OPTION_STORAGE & storage,  
    asio::error_code & ec);
```

5.181.3 serial_port_base::baud_rate::store

```
asio::error_code store(  
    ASIO_OPTION_STORAGE & storage,  
    asio::error_code & ec) const;
```

5.181.4 serial_port_base::baud_rate::value

```
unsigned int value() const;
```

5.182 serial_port_base::character_size

Serial port option to permit changing the character size.

```
class character_size
```

Member Functions

Name	Description
character_size	
load	
store	
value	

Implements changing the character size for a given serial port.

Requirements

Header: asio/serial_port_base.hpp

Convenience header: asio.hpp

5.182.1 serial_port_base::character_size::character_size

```
character_size(  
    unsigned int t = 8);
```

5.182.2 serial_port_base::character_size::load

```
asio::error_code load(
    const ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec);
```

5.182.3 serial_port_base::character_size::store

```
asio::error_code store(
    ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec) const;
```

5.182.4 serial_port_base::character_size::value

```
unsigned int value() const;
```

5.183 serial_port_base::flow_control

Serial port option to permit changing the flow control.

```
class flow_control
```

Types

Name	Description
type	

Member Functions

Name	Description
flow_control	
load	
store	
value	

Implements changing the flow control for a given serial port.

Requirements

Header: asio/serial_port_base.hpp

Convenience header: asio.hpp

5.183.1 serial_port_base::flow_control::flow_control

```
flow_control(
    type t = none);
```

5.183.2 serial_port_base::flow_control::load

```
asio::error_code load(
    const ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec);
```

5.183.3 serial_port_base::flow_control::store

```
asio::error_code store(
    ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec) const;
```

5.183.4 serial_port_base::flow_control::type

```
enum type
```

Values

none

software

hardware

5.183.5 serial_port_base::flow_control::value

```
type value() const;
```

5.184 serial_port_base::parity

Serial port option to permit changing the parity.

```
class parity
```

Types

Name	Description
type	

Member Functions

Name	Description
load	
parity	
store	
value	

Implements changing the parity for a given serial port.

Requirements

Header: asio/serial_port_base.hpp

Convenience header: asio.hpp

5.184.1 serial_port_base::parity::load

```
asio::error_code load(
    const ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec);
```

5.184.2 serial_port_base::parity::parity

```
parity(
    type t = none);
```

5.184.3 serial_port_base::parity::store

```
asio::error_code store(
    ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec) const;
```

5.184.4 serial_port_base::parity::type

```
enum type
```

Values

none

odd

even

5.184.5 serial_port_base::parity::value

```
type value() const;
```

5.185 serial_port_base::stop_bits

Serial port option to permit changing the number of stop bits.

```
class stop_bits
```

Types

Name	Description
type	

Member Functions

Name	Description
load	
stop_bits	
store	
value	

Implements changing the number of stop bits for a given serial port.

Requirements

Header: asio/serial_port_base.hpp

Convenience header: asio.hpp

5.185.1 serial_port_base::stop_bits::load

```
asio::error_code load(
    const ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec);
```

5.185.2 serial_port_base::stop_bits::stop_bits

```
stop_bits(
    type t = one);
```

5.185.3 serial_port_base::stop_bits::store

```
asio::error_code store(
    ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec) const;
```

5.185.4 serial_port_base::stop_bits::type

```
enum type
```

Values

one

onepointfive

two

5.185.5 `serial_port_base::stop_bits::value`

```
type value() const;
```

5.186 `serial_port_service`

Default service implementation for a serial port.

```
class serial_port_service :  
    public io_service::service
```

Types

Name	Description
<code>implementation_type</code>	The type of a serial port implementation.
<code>native_handle_type</code>	The native handle type.
<code>native_type</code>	(Deprecated: Use <code>native_handle_type</code> .) The native handle type.

Member Functions

Name	Description
<code>assign</code>	Assign an existing native handle to a serial port.
<code>async_read_some</code>	Start an asynchronous read.
<code>async_write_some</code>	Start an asynchronous write.
<code>cancel</code>	Cancel all asynchronous operations associated with the handle.
<code>close</code>	Close a serial port implementation.
<code>construct</code>	Construct a new serial port implementation.
<code>destroy</code>	Destroy a serial port implementation.
<code>get_io_service</code>	Get the <code>io_service</code> object that owns the service.
<code>get_option</code>	Get a serial port option.
<code>is_open</code>	Determine whether the handle is open.
<code>move_assign</code>	Move-assign from another serial port implementation.
<code>move_construct</code>	Move-construct a new serial port implementation.

Name	Description
native	(Deprecated: Use native_handle().) Get the native handle implementation.
native_handle	Get the native handle implementation.
open	Open a serial port.
read_some	Read some data from the stream.
send_break	Send a break sequence to the serial port.
serial_port_service	Construct a new serial port service for the specified io_service.
set_option	Set a serial port option.
write_some	Write the given data to the stream.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/serial_port_service.hpp

Convenience header: asio.hpp

5.186.1 serial_port_service::assign

Assign an existing native handle to a serial port.

```
asio::error_code assign(
    implementation_type & impl,
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.186.2 serial_port_service::async_read_some

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

5.186.3 serial_port_service::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

5.186.4 serial_port_service::cancel

Cancel all asynchronous operations associated with the handle.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.186.5 serial_port_service::close

Close a serial port implementation.

```
asio::error_code close(
    implementation_type & impl,
    asio::error_code & ec);
```

5.186.6 serial_port_service::construct

Construct a new serial port implementation.

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

5.186.7 serial_port_service::destroy

Destroy a serial port implementation.

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

5.186.8 serial_port_service::get_io_service

Inherited from io_service.

Get the **io_service** object that owns the service.

```
asio::io_service & get_io_service();
```

5.186.9 serial_port_service::get_option

Get a serial port option.

```
template<
    typename GettableSerialPortOption>
asio::error_code get_option(
    const implementation_type & impl,
    GettableSerialPortOption & option,
    asio::error_code & ec) const;
```

5.186.10 serial_port_service::id

The unique service identifier.

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

5.186.11 serial_port_service::implementation_type

The type of a serial port implementation.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asio/serial_port_service.hpp

Convenience header: asio.hpp

5.186.12 serial_port_service::is_open

Determine whether the handle is open.

```
bool is_open(
    const implementation_type & impl) const;
```

5.186.13 serial_port_service::move_assign

Move-assign from another serial port implementation.

```
void move_assign(
    implementation_type & impl,
    serial_port_service & other_service,
    implementation_type & other_impl);
```

5.186.14 serial_port_service::move_construct

Move-construct a new serial port implementation.

```
void move_construct(
    implementation_type & impl,
    implementation_type & other_impl);
```

5.186.15 serial_port_service::native

(Deprecated: Use native_handle().) Get the native handle implementation.

```
native_type native(
    implementation_type & impl);
```

5.186.16 serial_port_service::native_handle

Get the native handle implementation.

```
native_handle_type native_handle(
    implementation_type & impl);
```

5.186.17 serial_port_service::native_handle_type

The native handle type.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/serial_port_service.hpp

Convenience header: asio.hpp

5.186.18 serial_port_service::native_type

(Deprecated: Use native_handle_type.) The native handle type.

```
typedef implementation_defined native_type;
```

Requirements

Header: asio/serial_port_service.hpp

Convenience header: asio.hpp

5.186.19 serial_port_service::open

Open a serial port.

```
asio::error_code open(
    implementation_type & impl,
    const std::string & device,
    asio::error_code & ec);
```

5.186.20 serial_port_service::read_some

Read some data from the stream.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.186.21 serial_port_service::send_break

Send a break sequence to the serial port.

```
asio::error_code send_break(
    implementation_type & impl,
    asio::error_code & ec);
```

5.186.22 serial_port_service::serial_port_service

Construct a new serial port service for the specified [io_service](#).

```
serial_port_service(
    asio::io_service & io_service);
```

5.186.23 serial_port_service::set_option

Set a serial port option.

```
template<
    typename SettableSerialPortOption>
asio::error_code set_option(
    implementation_type & impl,
    const SettableSerialPortOption & option,
    asio::error_code & ec);
```

5.186.24 serial_port_service::write_some

Write the given data to the stream.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.187 service_already_exists

Exception thrown when trying to add a duplicate service to an [io_service](#).

```
class service_already_exists
```

Member Functions

Name	Description
service_already_exists	

Requirements

Header: `asio/io_service.hpp`

Convenience header: asio.hpp

5.187.1 service_already_exists::service_already_exists

```
service_already_exists();
```

5.188 signal_set

Typedef for the typical usage of a signal set.

```
typedef basic_signal_set signal_set;
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
add	Add a signal to a signal_set.
async_wait	Start an asynchronous operation to wait for a signal to be delivered.
basic_signal_set	Construct a signal set without adding any signals. Construct a signal set and add one signal. Construct a signal set and add two signals. Construct a signal set and add three signals.
cancel	Cancel all operations associated with the signal set.
clear	Remove all signals from a signal_set.
get_io_service	Get the io_service associated with the object.
remove	Remove a signal from a signal_set.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.

Name	Description
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_signal_set` class template provides the ability to perform an asynchronous wait for one or more signals to occur.

Most applications will use the `signal_set` typedef.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Example

Performing an asynchronous wait:

```
void handler(
    const asio::error_code& error,
    int signal_number)
{
    if (!error)
    {
        // A signal occurred.
    }
}

...

// Construct a signal set registered for process termination.
asio::signal_set signals(io_service, SIGINT, SIGTERM);

// Start an asynchronous wait for one of the signals to occur.
signals.async_wait(handler);
```

Queueing of signal notifications

If a signal is registered with a `signal_set`, and the signal occurs when there are no waiting handlers, then the signal notification is queued. The next `async_wait` operation on that `signal_set` will dequeue the notification. If multiple notifications are queued, subsequent `async_wait` operations dequeue them one at a time. Signal notifications are dequeued in order of ascending signal number.

If a signal number is removed from a `signal_set` (using the `remove` or `erase` member functions) then any queued notifications for that signal are discarded.

Multiple registration of signals

The same signal number may be registered with different signal_set objects. When the signal occurs, one handler is called for each signal_set object.

Note that multiple registration only works for signals that are registered using Asio. The application must not also register a signal handler using functions such as `signal()` or `sigaction()`.

Signal masking on POSIX platforms

POSIX allows signals to be blocked using functions such as `sigprocmask()` and `pthread_sigmask()`. For signals to be delivered, programs must ensure that any signals registered using signal_set objects are unblocked in at least one thread.

Requirements

Header: `asio/signal_set.hpp`

Convenience header: `asio.hpp`

5.189 signal_set_service

Default service implementation for a signal set.

```
class signal_set_service :  
    public io_service::service
```

Types

Name	Description
implementation_type	The type of a signal set implementation.

Member Functions

Name	Description
add	Add a signal to a signal_set.
async_wait	
cancel	Cancel all operations associated with the signal set.
clear	Remove all signals from a signal_set.
construct	Construct a new signal set implementation.
destroy	Destroy a signal set implementation.
get_io_service	Get the io_service object that owns the service.
remove	Remove a signal to a signal_set.

Name	Description
signal_set_service	Construct a new signal set service for the specified io_service.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/signal_set_service.hpp

Convenience header: asio.hpp

5.189.1 signal_set_service::add

Add a signal to a signal_set.

```
asio::error_code add(
    implementation_type & impl,
    int signal_number,
    asio::error_code & ec);
```

5.189.2 signal_set_service::async_wait

```
template<
    typename SignalHandler>
void-or-deduced async_wait(
    implementation_type & impl,
    SignalHandler handler);
```

5.189.3 signal_set_service::cancel

Cancel all operations associated with the signal set.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.189.4 signal_set_service::clear

Remove all signals from a signal_set.

```
asio::error_code clear(
    implementation_type & impl,
    asio::error_code & ec);
```

5.189.5 signal_set_service::construct

Construct a new signal set implementation.

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

5.189.6 signal_set_service::destroy

Destroy a signal set implementation.

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

5.189.7 signal_set_service::get_io_service

Inherited from io_service.

Get the **io_service** object that owns the service.

```
asio::io_service & get_io_service();
```

5.189.8 signal_set_service::id

The unique service identifier.

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

5.189.9 signal_set_service::implementation_type

The type of a signal set implementation.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asio/signal_set_service.hpp

Convenience header: asio.hpp

5.189.10 signal_set_service::remove

Remove a signal to a signal_set.

```
asio::error_code remove(  
    implementation_type & impl,  
    int signal_number,  
    asio::error_code & ec);
```

5.189.11 signal_set_service::signal_set_service

Construct a new signal set service for the specified `io_service`.

```
signal_set_service(
   asio::io_service & io_service);
```

5.190 socket_acceptor_service

Default service implementation for a socket acceptor.

```
template<
    typename Protocol>
class socket_acceptor_service :
    public io_service::service
```

Types

Name	Description
<code>endpoint_type</code>	The endpoint type.
<code>implementation_type</code>	The native type of the socket acceptor.
<code>native_handle_type</code>	The native acceptor type.
<code>native_type</code>	(Deprecated: Use <code>native_handle_type</code> .) The native acceptor type.
<code>protocol_type</code>	The protocol type.

Member Functions

Name	Description
<code>accept</code>	Accept a new connection.
<code>assign</code>	Assign an existing native acceptor to a socket acceptor.
<code>async_accept</code>	Start an asynchronous accept.
<code>bind</code>	Bind the socket acceptor to the specified local endpoint.
<code>cancel</code>	Cancel all asynchronous operations associated with the acceptor.
<code>close</code>	Close a socket acceptor implementation.
<code>construct</code>	Construct a new socket acceptor implementation.
<code>converting_move_construct</code>	Move-construct a new socket acceptor implementation from another protocol type.

Name	Description
destroy	Destroy a socket acceptor implementation.
get_io_service	Get the io_service object that owns the service.
get_option	Get a socket option.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the acceptor is open.
listen	Place the socket acceptor into the state where it will listen for new connections.
local_endpoint	Get the local endpoint.
move_assign	Move-assign from another socket acceptor implementation.
move_construct	Move-construct a new socket acceptor implementation.
native	(Deprecated: Use native_handle().) Get the native acceptor implementation.
native_handle	Get the native acceptor implementation.
native_non_blocking	Gets the non-blocking mode of the native acceptor implementation. Sets the non-blocking mode of the native acceptor implementation.
non_blocking	Gets the non-blocking mode of the acceptor. Sets the non-blocking mode of the acceptor.
open	Open a new socket acceptor implementation.
set_option	Set a socket option.
socket_acceptor_service	Construct a new socket acceptor service for the specified io_service.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/socket_acceptor_service.hpp

Convenience header: asio.hpp

5.190.1 socket_acceptor_service::accept

Accept a new connection.

```
template<
    typename Protocol1,
    typename SocketService>
asio::error_code accept(
    implementation_type & impl,
    basic_socket< Protocol1, SocketService > & peer,
    endpoint_type * peer_endpoint,
    asio::error_code & ec,
    typename enable_if< is_convertible< Protocol, Protocol1 >::value >::type * = 0);
```

5.190.2 socket_acceptor_service::assign

Assign an existing native acceptor to a socket acceptor.

```
asio::error_code assign(
    implementation_type & impl,
    const protocol_type & protocol,
    const native_handle_type & native_acceptor,
    asio::error_code & ec);
```

5.190.3 socket_acceptor_service::async_accept

Start an asynchronous accept.

```
template<
    typename Protocol1,
    typename SocketService,
    typename AcceptHandler>
void-or-deduced async_accept(
    implementation_type & impl,
    basic_socket< Protocol1, SocketService > & peer,
    endpoint_type * peer_endpoint,
    AcceptHandler handler,
    typename enable_if< is_convertible< Protocol, Protocol1 >::value >::type * = 0);
```

5.190.4 socket_acceptor_service::bind

Bind the socket acceptor to the specified local endpoint.

```
asio::error_code bind(
    implementation_type & impl,
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.190.5 socket_acceptor_service::cancel

Cancel all asynchronous operations associated with the acceptor.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.190.6 `socket_acceptor_service::close`

Close a socket acceptor implementation.

```
asio::error_code close(
    implementation_type & impl,
    asio::error_code & ec);
```

5.190.7 `socket_acceptor_service::construct`

Construct a new socket acceptor implementation.

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

5.190.8 `socket_acceptor_service::converting_move_construct`

Move-construct a new socket acceptor implementation from another protocol type.

```
template<
    typename Protocol1>
void converting_move_construct(
    implementation_type & impl,
    typename socket_acceptor_service< Protocol1 >::implementation_type & other_impl,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.190.9 `socket_acceptor_service::destroy`

Destroy a socket acceptor implementation.

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

5.190.10 `socket_acceptor_service::endpoint_type`

The endpoint type.

```
typedef protocol_type::endpoint endpoint_type;
```

Requirements

Header: asio/socket_acceptor_service.hpp

Convenience header: asio.hpp

5.190.11 `socket_acceptor_service::get_io_service`

Inherited from io_service.

Get the `io_service` object that owns the service.

```
asio::io_service & get_io_service();
```

5.190.12 socket_acceptor_service::get_option

Get a socket option.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    const implementation_type & impl,
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.190.13 socket_acceptor_service::id

The unique service identifier.

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

5.190.14 socket_acceptor_service::implementation_type

The native type of the socket acceptor.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asio/socket_acceptor_service.hpp

Convenience header: asio.hpp

5.190.15 socket_acceptor_service::io_control

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    implementation_type & impl,
    IoControlCommand & command,
    asio::error_code & ec);
```

5.190.16 socket_acceptor_service::is_open

Determine whether the acceptor is open.

```
bool is_open(
    const implementation_type & impl) const;
```

5.190.17 socket_acceptor_service::listen

Place the socket acceptor into the state where it will listen for new connections.

```
asio::error_code listen(
    implementation_type & impl,
    int backlog,
    asio::error_code & ec);
```

5.190.18 `socket_acceptor_service::local_endpoint`

Get the local endpoint.

```
endpoint_type local_endpoint(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.190.19 `socket_acceptor_service::move_assign`

Move-assign from another socket acceptor implementation.

```
void move_assign(
    implementation_type & impl,
    socket_acceptor_service & other_service,
    implementation_type & other_impl);
```

5.190.20 `socket_acceptor_service::move_construct`

Move-construct a new socket acceptor implementation.

```
void move_construct(
    implementation_type & impl,
    implementation_type & other_impl);
```

5.190.21 `socket_acceptor_service::native`

(Deprecated: Use `native_handle()`.) Get the native acceptor implementation.

```
native_type native(
    implementation_type & impl);
```

5.190.22 `socket_acceptor_service::native_handle`

Get the native acceptor implementation.

```
native_handle_type native_handle(
    implementation_type & impl);
```

5.190.23 `socket_acceptor_service::native_handle_type`

The native acceptor type.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/socket_acceptor_service.hpp

Convenience header: asio.hpp

5.190.24 `socket_acceptor_service::native_non_blocking`

Gets the non-blocking mode of the native acceptor implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the native acceptor implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.190.24.1 `socket_acceptor_service::native_non_blocking (1 of 2 overloads)`

Gets the non-blocking mode of the native acceptor implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

5.190.24.2 `socket_acceptor_service::native_non_blocking (2 of 2 overloads)`

Sets the non-blocking mode of the native acceptor implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.190.25 `socket_acceptor_service::native_type`

(Deprecated: Use `native_handle_type`.) The native acceptor type.

```
typedef implementation_defined native_type;
```

Requirements

Header: `asio/socket_acceptor_service.hpp`

Convenience header: `asio.hpp`

5.190.26 `socket_acceptor_service::non_blocking`

Gets the non-blocking mode of the acceptor.

```
bool non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the acceptor.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.190.26.1 socket_acceptor_service::non_blocking (1 of 2 overloads)

Gets the non-blocking mode of the acceptor.

```
bool non_blocking(
    const implementation_type & impl) const;
```

5.190.26.2 socket_acceptor_service::non_blocking (2 of 2 overloads)

Sets the non-blocking mode of the acceptor.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.190.27 socket_acceptor_service::open

Open a new socket acceptor implementation.

```
asio::error_code open(
    implementation_type & impl,
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.190.28 socket_acceptor_service::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/socket_acceptor_service.hpp

Convenience header: asio.hpp

5.190.29 socket_acceptor_service::set_option

Set a socket option.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    implementation_type & impl,
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.190.30 socket_acceptor_service::socket_acceptor_service

Construct a new socket acceptor service for the specified [io_service](#).

```
socket_acceptor_service(
    asio::io_service & io_service);
```

5.191 socket_base

The `socket_base` class is used as a base for the `basic_stream_socket` and `basic_datagram_socket` class templates so that we have a common place to define the `shutdown_type` and enum.

```
class socket_base
```

Types

Name	Description
<code>broadcast</code>	Socket option to permit sending of broadcast messages.
<code>bytes_readable</code>	IO control command to get the amount of data that can be read without blocking.
<code>debug</code>	Socket option to enable socket-level debugging.
<code>do_not_route</code>	Socket option to prevent routing, use local interfaces only.
<code>enable_connection_aborted</code>	Socket option to report aborted connections on accept.
<code>keep_alive</code>	Socket option to send keep-alives.
<code>linger</code>	Socket option to specify whether the socket lingers on close if unsent data is present.
<code>message_flags</code>	Bitmask type for flags that can be passed to send and receive operations.
<code>non_blocking_io</code>	(Deprecated: Use <code>non_blocking()</code> .) IO control command to set the blocking mode of the socket.
<code>receive_buffer_size</code>	Socket option for the receive buffer size of a socket.
<code>receive_low_watermark</code>	Socket option for the receive low watermark.
<code>reuse_address</code>	Socket option to allow the socket to be bound to an address that is already in use.
<code>send_buffer_size</code>	Socket option for the send buffer size of a socket.
<code>send_low_watermark</code>	Socket option for the send low watermark.
<code>shutdown_type</code>	Different ways a socket may be shutdown.

Protected Member Functions

Name	Description
<code>~socket_base</code>	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	The maximum length of the queue of pending incoming connections.
message_do_not_route	Specify that the data should not be subject to routing.
message_end_of_record	Specifies that the data marks the end of a record.
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.1 socket_base::broadcast

Socket option to permit sending of broadcast messages.

```
typedef implementation_defined broadcast;
```

Implements the SOL_SOCKET/SO_BROADCAST socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::broadcast option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.2 `socket_base::bytes_readable`

IO control command to get the amount of data that can be read without blocking.

```
typedef implementation_defined bytes_readable;
```

Implements the FIONREAD IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::bytes_readable command(true);
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.3 `socket_base::debug`

Socket option to enable socket-level debugging.

```
typedef implementation_defined debug;
```

Implements the SOL_SOCKET/SO_DEBUG socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::debug option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.4 `socket_base::do_not_route`

Socket option to prevent routing, use local interfaces only.

```
typedef implementation_defined do_not_route;
```

Implements the SOL_SOCKET/SO_DONTROUTE socket option.

Examples

Setting the option:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_service);
...
asio::socket_base::do_not_route option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.5 `socket_base::enable_connection_aborted`

Socket option to report aborted connections on accept.

```
typedef implementation_defined enable_connection_aborted;
```

Implements a custom socket option that determines whether or not an accept operation is permitted to fail with `asio::error::connection_aborted`. By default the option is false.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::enable_connection_aborted option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.6 socket_base::keep_alive

Socket option to send keep-alives.

```
typedef implementation_defined keep_alive;
```

Implements the SOL_SOCKET/SO_KEEPALIVE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.7 socket_base::linger

Socket option to specify whether the socket lingers on close if unsent data is present.

```
typedef implementation_defined linger;
```

Implements the SOL_SOCKET/SO_LINGER socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::linger option;
socket.get_option(option);
bool is_set = option.enabled();
unsigned short timeout = option.timeout();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.8 socket_base::max_connections

The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.191.9 socket_base::message_do_not_route

Specify that the data should not be subject to routing.

```
static const int message_do_not_route = implementation_defined;
```

5.191.10 socket_base::message_end_of_record

Specifies that the data marks the end of a record.

```
static const int message_end_of_record = implementation_defined;
```

5.191.11 socket_base::message_flags

Bitmask type for flags that can be passed to send and receive operations.

```
typedef int message_flags;
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.12 socket_base::message_out_of_band

Process out-of-band data.

```
static const int message_out_of_band = implementation_defined;
```

5.191.13 `socket_base::message_peek`

Peek at incoming data without removing it from the input queue.

```
static const int message_peek = implementation_defined;
```

5.191.14 `socket_base::non_blocking_io`

(Deprecated: Use `non_blocking()`) IO control command to set the blocking mode of the socket.

```
typedef implementation_defined non_blocking_io;
```

Implements the FIONBIO IO control command.

Example

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::non_blocking_io command(true);
socket.io_control(command);
```

Requirements

Header: `asio/socket_base.hpp`

Convenience header: `asio.hpp`

5.191.15 `socket_base::receive_buffer_size`

Socket option for the receive buffer size of a socket.

```
typedef implementation_defined receive_buffer_size;
```

Implements the SOL_SOCKET/SO_RCVBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: `asio/socket_base.hpp`

Convenience header: `asio.hpp`

5.191.16 socket_base::receive_low_watermark

Socket option for the receive low watermark.

```
typedef implementation_defined receive_low_watermark;
```

Implements the SOL_SOCKET/SO_RCVLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::receive_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.17 socket_base::reuse_address

Socket option to allow the socket to be bound to an address that is already in use.

```
typedef implementation_defined reuse_address;
```

Implements the SOL_SOCKET/SO_REUSEADDR socket option.

Examples

Setting the option:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_service);
...
asio::socket_base::reuse_address option;
acceptor.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.18 socket_base::send_buffer_size

Socket option for the send buffer size of a socket.

```
typedef implementation_defined send_buffer_size;
```

Implements the SOL_SOCKET/SO_SNDBUF socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_buffer_size option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.19 socket_base::send_low_watermark

Socket option for the send low watermark.

```
typedef implementation_defined send_low_watermark;
```

Implements the SOL_SOCKET/SO SNDLOWAT socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_service);
...
asio::socket_base::send_low_watermark option;
socket.get_option(option);
int size = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.191.20 socket_base::shutdown_type

Different ways a socket may be shutdown.

```
enum shutdown_type
```

Values

shutdown_receive Shutdown the receive side of the socket.

shutdown_send Shutdown the send side of the socket.

shutdown_both Shutdown both send and receive on the socket.

5.191.21 socket_base::~socket_base

Protected destructor to prevent deletion through this type.

```
~socket_base();
```

5.192 spawn

Start a new stackful coroutine.

```
template<
    typename Handler,
    typename Function>
void spawn(
    Handler handler,
    Function function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());

template<
    typename Handler,
    typename Function>
void spawn(
    basic_yield_context<Handler> ctx,
    Function function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());

template<
    typename Function>
void spawn(
    asio::io_service::strand strand,
    Function function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());

template<
```

```

    typename Function>
void spawn(
    asio::io_service & io_service,
    Function function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());

```

The `spawn()` function is a high-level wrapper over the Boost.Coroutine library. This function enables programs to implement asynchronous logic in a synchronous manner, as illustrated by the following example:

```

asio::spawn(my_strand, do_echo);

// ...

void do_echo(asio::yield_context yield)
{
    try
    {
        char data[128];
        for (;;)
        {
            std::size_t length =
                my_socket.async_read_some(
                    asio::buffer(data), yield);

            asio::async_write(my_socket,
                asio::buffer(data, length), yield);
        }
    }
    catch (std::exception& e)
    {
        // ...
    }
}

```

Requirements

Header: `asio/spawn.hpp`

Convenience header: None

5.192.1 `spawn` (1 of 4 overloads)

Start a new stackful coroutine, calling the specified handler when it completes.

```

template<
    typename Handler,
    typename Function>
void spawn(
    Handler handler,
    Function function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());

```

This function is used to launch a new coroutine.

Parameters

handler A handler to be called when the coroutine exits. More importantly, the handler provides an execution context (via the the handler invocation hook) for the coroutine. The handler must have the signature:

```
void handler();
```

function The coroutine function. The function must have the signature:

```
void function(basic_yield_context<Handler> yield);
```

attributes Boost.Coroutine attributes used to customise the coroutine.

5.192.2 spawn (2 of 4 overloads)

Start a new stackful coroutine, inheriting the execution context of another.

```
template<
    typename Handler,
    typename Function>
void spawn(
    basic_yield_context< Handler > ctx,
    Function function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());
```

This function is used to launch a new coroutine.

Parameters

ctx Identifies the current coroutine as a parent of the new coroutine. This specifies that the new coroutine should inherit the execution context of the parent. For example, if the parent coroutine is executing in a particular strand, then the new coroutine will execute in the same strand.

function The coroutine function. The function must have the signature:

```
void function(basic_yield_context<Handler> yield);
```

attributes Boost.Coroutine attributes used to customise the coroutine.

5.192.3 spawn (3 of 4 overloads)

Start a new stackful coroutine that executes in the context of a strand.

```
template<
    typename Function>
void spawn(
    asio::io_service::strand strand,
    Function function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());
```

This function is used to launch a new coroutine.

Parameters

strand Identifies a strand. By starting multiple coroutines on the same strand, the implementation ensures that none of those coroutines can execute simultaneously.

function The coroutine function. The function must have the signature:

```
void function(yield_context yield);
```

attributes Boost.Coroutine attributes used to customise the coroutine.

5.192.4 spawn (4 of 4 overloads)

Start a new stackful coroutine that executes on a given `io_service`.

```
template<
    typename Function>
void spawn(
    asio::io_service & io_service,
    Function function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());
```

This function is used to launch a new coroutine.

Parameters

io_service Identifies the `io_service` that will run the coroutine. The new coroutine is implicitly given its own strand within this `io_service`.

function The coroutine function. The function must have the signature:

```
void function(yield_context yield);
```

attributes Boost.Coroutine attributes used to customise the coroutine.

5.193 ssl::context

```
class context :
public ssl::context_base,
noncopyable
```

Types

Name	Description
<code>file_format</code>	File format types.
<code>impl_type</code>	(Deprecated: Use <code>native_handle_type</code> .) The native type of the SSL context.
<code>method</code>	Different methods supported by a context.
<code>native_handle_type</code>	The native handle type of the SSL context.
<code>options</code>	Bitmask type for SSL options.
<code>password_purpose</code>	Purpose of PEM password.

Member Functions

Name	Description
<code>add_certificate_authority</code>	Add certification authority for performing verification.
<code>add_verify_path</code>	Add a directory containing certificate authority files to be used for performing verification.
<code>clear_options</code>	Clear options on the context.
<code>context</code>	Constructor. Deprecated constructor taking a reference to an <code>io_service</code> object. Move-construct a context from another.
<code>impl</code>	(Deprecated: Use <code>native_handle()</code>) Get the underlying implementation in the native type.
<code>load_verify_file</code>	Load a certification authority file for performing verification.
<code>native_handle</code>	Get the underlying implementation in the native type.
<code>operator=</code>	Move-assign a context from another.
<code>set_default_verify_paths</code>	Configures the context to use the default directories for finding certification authority certificates.
<code>set_options</code>	Set options on the context.
<code>set_password_callback</code>	Set the password callback.
<code>set_verify_callback</code>	Set the callback used to verify peer certificates.
<code>set_verify_depth</code>	Set the peer verification depth.
<code>set_verify_mode</code>	Set the peer verification mode.
<code>use_certificate</code>	Use a certificate from a memory buffer.
<code>use_certificate_chain</code>	Use a certificate chain from a memory buffer.
<code>use_certificate_chain_file</code>	Use a certificate chain from a file.
<code>use_certificate_file</code>	Use a certificate from a file.
<code>use_private_key</code>	Use a private key from a memory buffer.
<code>use_private_key_file</code>	Use a private key from a file.
<code>use_rsa_private_key</code>	Use an RSA private key from a memory buffer.
<code>use_rsa_private_key_file</code>	Use an RSA private key from a file.
<code>use_tmp_dh</code>	Use the specified memory buffer to obtain the temporary Diffie-Hellman parameters.

Name	Description
use_tmp_dh_file	Use the specified file to obtain the temporary Diffie-Hellman parameters.
~context	Destructor.

Data Members

Name	Description
default_workarounds	Implement various bug workarounds.
no_compression	Disable compression. Compression is disabled by default.
no_sslv2	Disable SSL v2.
no_sslv3	Disable SSL v3.
no_tlsv1	Disable TLS v1.
no_tlsv1_1	Disable TLS v1.1.
no_tlsv1_2	Disable TLS v1.2.
single_dh_use	Always create a new key when using tmp_dh parameters.

Requirements

Header: asio/ssl/context.hpp

Convenience header: asio/ssl.hpp

5.193.1 ssl::context::add_certificate_authority

Add certification authority for performing verification.

```
void add_certificate_authority(
    const const_buffer & ca);

asio::error_code add_certificate_authority(
    const const_buffer & ca,
    asio::error_code & ec);
```

5.193.1.1 ssl::context::add_certificate_authority (1 of 2 overloads)

Add certification authority for performing verification.

```
void add_certificate_authority(
    const const_buffer & ca);
```

This function is used to add one trusted certification authority from a memory buffer.

Parameters

ca The buffer containing the certification authority certificate. The certificate must use the PEM format.

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_CTX_get_cert_store and X509_STORE_add_cert.

5.193.1.2 `ssl::context::add_certificate_authority` (2 of 2 overloads)

Add certification authority for performing verification.

```
asio::error_code add_certificate_authority(
    const const_buffer & ca,
    asio::error_code & ec);
```

This function is used to add one trusted certification authority from a memory buffer.

Parameters

ca The buffer containing the certification authority certificate. The certificate must use the PEM format.

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_get_cert_store and X509_STORE_add_cert.

5.193.2 `ssl::context::add_verify_path`

Add a directory containing certificate authority files to be used for performing verification.

```
void add_verify_path(
    const std::string & path);

asio::error_code add_verify_path(
    const std::string & path,
    asio::error_code & ec);
```

5.193.2.1 `ssl::context::add_verify_path` (1 of 2 overloads)

Add a directory containing certificate authority files to be used for performing verification.

```
void add_verify_path(
    const std::string & path);
```

This function is used to specify the name of a directory containing certification authority certificates. Each file in the directory must contain a single certificate. The files must be named using the subject name's hash and an extension of ".0".

Parameters

path The name of a directory containing the certificates.

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_CTX_load_verify_locations.

5.193.2.2 ssl::context::add_verify_path (2 of 2 overloads)

Add a directory containing certificate authority files to be used for performing verification.

```
asio::error_code add_verify_path(
    const std::string & path,
    asio::error_code & ec);
```

This function is used to specify the name of a directory containing certification authority certificates. Each file in the directory must contain a single certificate. The files must be named using the subject name's hash and an extension of ".0".

Parameters

path The name of a directory containing the certificates.

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_load_verify_locations.

5.193.3 ssl::context::clear_options

Clear options on the context.

```
void clear_options(
    options o);

asio::error_code clear_options(
    options o,
    asio::error_code & ec);
```

5.193.3.1 ssl::context::clear_options (1 of 2 overloads)

Clear options on the context.

```
void clear_options(
    options o);
```

This function may be used to configure the SSL options used by the context.

Parameters

- o A bitmask of options. The available option values are defined in the `ssl::context_base` class. The specified options, if currently enabled on the context, are cleared.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_CTX_clear_options`.

5.193.3.2 `ssl::context::clear_options` (2 of 2 overloads)

Clear options on the context.

```
asio::error_code clear_options(
    options o,
    asio::error_code & ec);
```

This function may be used to configure the SSL options used by the context.

Parameters

- o A bitmask of options. The available option values are defined in the `ssl::context_base` class. The specified options, if currently enabled on the context, are cleared.
- ec Set to indicate what error occurred, if any.

Remarks

Calls `SSL_CTX_clear_options`.

5.193.4 `ssl::context::context`

Constructor.

```
explicit context(
    method m);
```

Deprecated constructor taking a reference to an `io_service` object.

```
context(
    asio::io_service & ,
    method m);
```

Move-construct a context from another.

```
context(
    context && other);
```

5.193.4.1 `ssl::context::context (1 of 3 overloads)`

Constructor.

```
context(  
    method m);
```

5.193.4.2 `ssl::context::context (2 of 3 overloads)`

Deprecated constructor taking a reference to an `io_service` object.

```
context(  
    asio::io_service & ,  
    method m);
```

5.193.4.3 `ssl::context::context (3 of 3 overloads)`

Move-construct a context from another.

```
context(  
    context && other);
```

This constructor moves an SSL context from one object to another.

Parameters

other The other context object from which the move will occur.

Remarks

Following the move, the following operations only are valid for the moved-from object: * Destruction.

- As a target for move-assignment.

5.193.5 `ssl::context::default_workarounds`

Implement various bug workarounds.

```
static const long default_workarounds = implementation_defined;
```

5.193.6 `ssl::context::file_format`

File format types.

```
enum file_format
```

Values

asn1 ASN.1 file.

pem PEM file.

5.193.7 `ssl::context::impl`

(Deprecated: Use `native_handle()`.) Get the underlying implementation in the native type.

```
impl_type impl();
```

This function may be used to obtain the underlying implementation of the context. This is intended to allow access to context functionality that is not otherwise provided.

5.193.8 `ssl::context::impl_type`

(Deprecated: Use `native_handle_type()`.) The native type of the SSL context.

```
typedef SSL_CTX * impl_type;
```

Requirements

Header: `asio/ssl/context.hpp`

Convenience header: `asio/ssl.hpp`

5.193.9 `ssl::context::load_verify_file`

Load a certification authority file for performing verification.

```
void load_verify_file(
    const std::string & filename);

asio::error_code load_verify_file(
    const std::string & filename,
    asio::error_code & ec);
```

5.193.9.1 `ssl::context::load_verify_file (1 of 2 overloads)`

Load a certification authority file for performing verification.

```
void load_verify_file(
    const std::string & filename);
```

This function is used to load one or more trusted certification authorities from a file.

Parameters

filename The name of a file containing certification authority certificates in PEM format.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_CTX_load_verify_locations`.

5.193.9.2 `ssl::context::load_verify_file` (2 of 2 overloads)

Load a certification authority file for performing verification.

```
asio::error_code load_verify_file(
    const std::string & filename,
    asio::error_code & ec);
```

This function is used to load the certificates for one or more trusted certification authorities from a file.

Parameters

filename The name of a file containing certification authority certificates in PEM format.

ec Set to indicate what error occurred, if any.

Remarks

Calls `SSL_CTX_load_verify_locations`.

5.193.10 `ssl::context::method`

Different methods supported by a context.

```
enum method
```

Values

sslv2 Generic SSL version 2.

sslv2_client SSL version 2 client.

sslv2_server SSL version 2 server.

sslv3 Generic SSL version 3.

sslv3_client SSL version 3 client.

sslv3_server SSL version 3 server.

tlsv1 Generic TLS version 1.

tlsv1_client TLS version 1 client.

tlsv1_server TLS version 1 server.

sslv23 Generic SSL/TLS.

sslv23_client SSL/TLS client.

sslv23_server SSL/TLS server.

tlsv11 Generic TLS version 1.1.

tlsv11_client TLS version 1.1 client.

tlsv11_server TLS version 1.1 server.

tlsv12 Generic TLS version 1.2.

tlsv12_client TLS version 1.2 client.

tlsv12_server TLS version 1.2 server.

5.193.11 `ssl::context::native_handle`

Get the underlying implementation in the native type.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying implementation of the context. This is intended to allow access to context functionality that is not otherwise provided.

5.193.12 `ssl::context::native_handle_type`

The native handle type of the SSL context.

```
typedef SSL_CTX * native_handle_type;
```

Requirements

Header: `asio/ssl/context.hpp`

Convenience header: `asio/ssl.hpp`

5.193.13 `ssl::context::no_compression`

Disable compression. Compression is disabled by default.

```
static const long no_compression = implementation_defined;
```

5.193.14 `ssl::context::no_sslv2`

Disable SSL v2.

```
static const long no_sslv2 = implementation_defined;
```

5.193.15 `ssl::context::no_sslv3`

Disable SSL v3.

```
static const long no_sslv3 = implementation_defined;
```

5.193.16 `ssl::context::no_tlsv1`

Disable TLS v1.

```
static const long no_tlsv1 = implementation_defined;
```

5.193.17 `ssl::context::no_tlsv1_1`

Disable TLS v1.1.

```
static const long no_tlsv1_1 = implementation_defined;
```

5.193.18 `ssl::context::no_tls1_2`

Disable TLS v1.2.

```
static const long no_tls1_2 = implementation_defined;
```

5.193.19 `ssl::context::operator=`

Move-assign a context from another.

```
context & operator=(  
    context && other);
```

This assignment operator moves an SSL context from one object to another.

Parameters

other The other context object from which the move will occur.

Remarks

Following the move, the following operations only are valid for the moved-from object: * Destruction.

- As a target for move-assignment.

5.193.20 `ssl::context::options`

Bitmask type for SSL options.

```
typedef long options;
```

Requirements

Header: asio/ssl/context.hpp

Convenience header: asio/ssl.hpp

5.193.21 `ssl::context::password_purpose`

Purpose of PEM password.

```
enum password_purpose
```

Values

for_reading The password is needed for reading/decryption.

for_writing The password is needed for writing/encryption.

5.193.22 `ssl::context::set_default_verify_paths`

Configures the context to use the default directories for finding certification authority certificates.

```
void set_default_verify_paths();

asio::error_code set_default_verify_paths(
    asio::error_code & ec);
```

5.193.22.1 `ssl::context::set_default_verify_paths (1 of 2 overloads)`

Configures the context to use the default directories for finding certification authority certificates.

```
void set_default_verify_paths();
```

This function specifies that the context should use the default, system-dependent directories for locating certification authority certificates.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_CTX_set_default_verify_paths`.

5.193.22.2 `ssl::context::set_default_verify_paths (2 of 2 overloads)`

Configures the context to use the default directories for finding certification authority certificates.

```
asio::error_code set_default_verify_paths(
    asio::error_code & ec);
```

This function specifies that the context should use the default, system-dependent directories for locating certification authority certificates.

Parameters

`ec` Set to indicate what error occurred, if any.

Remarks

Calls `SSL_CTX_set_default_verify_paths`.

5.193.23 `ssl::context::set_options`

Set options on the context.

```
void set_options(
    options o);

asio::error_code set_options(
    options o,
    asio::error_code & ec);
```

5.193.23.1 `ssl::context::set_options` (1 of 2 overloads)

Set options on the context.

```
void set_options(  
    options o);
```

This function may be used to configure the SSL options used by the context.

Parameters

- o A bitmask of options. The available option values are defined in the `ssl::context_base` class. The options are bitwise-ored with any existing value for the options.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_CTX_set_options`.

5.193.23.2 `ssl::context::set_options` (2 of 2 overloads)

Set options on the context.

```
asio::error_code set_options(  
    options o,  
    asio::error_code & ec);
```

This function may be used to configure the SSL options used by the context.

Parameters

- o A bitmask of options. The available option values are defined in the `ssl::context_base` class. The options are bitwise-ored with any existing value for the options.
- ec Set to indicate what error occurred, if any.

Remarks

Calls `SSL_CTX_set_options`.

5.193.24 `ssl::context::set_password_callback`

Set the password callback.

```
template<  
    typename PasswordCallback>  
void set_password_callback(  
    PasswordCallback callback);  
  
template<  
    typename PasswordCallback>  
asio::error_code set_password_callback(  
    PasswordCallback callback,  
    asio::error_code & ec);
```

5.193.24.1 `ssl::context::set_password_callback` (1 of 2 overloads)

Set the password callback.

```
template<
    typename PasswordCallback>
void set_password_callback(
    PasswordCallback callback);
```

This function is used to specify a callback function to obtain password information about an encrypted key in PEM format.

Parameters

callback The function object to be used for obtaining the password. The function signature of the handler must be:

```
std::string password_callback(
    std::size_t max_length, // The maximum size for a password.
    password_purpose purpose // Whether password is for reading or writing.
);
```

The return value of the callback is a string containing the password.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_CTX_set_default_passwd_cb`.

5.193.24.2 `ssl::context::set_password_callback` (2 of 2 overloads)

Set the password callback.

```
template<
    typename PasswordCallback>
asio::error_code set_password_callback(
    PasswordCallback callback,
    asio::error_code & ec);
```

This function is used to specify a callback function to obtain password information about an encrypted key in PEM format.

Parameters

callback The function object to be used for obtaining the password. The function signature of the handler must be:

```
std::string password_callback(
    std::size_t max_length, // The maximum size for a password.
    password_purpose purpose // Whether password is for reading or writing.
);
```

The return value of the callback is a string containing the password.

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_set_default_passwd_cb.

5.193.25 ssl::context::set_verify_callback

Set the callback used to verify peer certificates.

```
template<
    typename VerifyCallback>
void set_verify_callback(
    VerifyCallback callback);

template<
    typename VerifyCallback>
asio::error_code set_verify_callback(
    VerifyCallback callback,
    asio::error_code & ec);
```

5.193.25.1 ssl::context::set_verify_callback (1 of 2 overloads)

Set the callback used to verify peer certificates.

```
template<
    typename VerifyCallback>
void set_verify_callback(
    VerifyCallback callback);
```

This function is used to specify a callback function that will be called by the implementation when it needs to verify a peer certificate.

Parameters

callback The function object to be used for verifying a certificate. The function signature of the handler must be:

```
bool verify_callback(
    bool preverified, // True if the certificate passed pre-verification.
    verify_context& ctx // The peer certificate and other context.
);
```

The return value of the callback is true if the certificate has passed verification, false otherwise.

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_CTX_set_verify.

5.193.25.2 `ssl::context::set_verify_callback` (2 of 2 overloads)

Set the callback used to verify peer certificates.

```
template<
    typename VerifyCallback>
asio::error_code set_verify_callback(
    VerifyCallback callback,
    asio::error_code & ec);
```

This function is used to specify a callback function that will be called by the implementation when it needs to verify a peer certificate.

Parameters

callback The function object to be used for verifying a certificate. The function signature of the handler must be:

```
bool verify_callback(
    bool preverified, // True if the certificate passed pre-verification.
    verify_context& ctx // The peer certificate and other context.
);
```

The return value of the callback is true if the certificate has passed verification, false otherwise.

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_set_verify.

5.193.26 `ssl::context::set_verify_depth`

Set the peer verification depth.

```
void set_verify_depth(
    int depth);

asio::error_code set_verify_depth(
    int depth,
    asio::error_code & ec);
```

5.193.26.1 `ssl::context::set_verify_depth` (1 of 2 overloads)

Set the peer verification depth.

```
void set_verify_depth(
    int depth);
```

This function may be used to configure the maximum verification depth allowed by the context.

Parameters

depth Maximum depth for the certificate chain verification that shall be allowed.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_CTX_set_verify_depth`.

5.193.26.2 `ssl::context::set_verify_depth (2 of 2 overloads)`

Set the peer verification depth.

```
asio::error_code set_verify_depth(
    int depth,
    asio::error_code & ec);
```

This function may be used to configure the maximum verification depth allowed by the context.

Parameters

depth Maximum depth for the certificate chain verification that shall be allowed.

ec Set to indicate what error occurred, if any.

Remarks

Calls `SSL_CTX_set_verify_depth`.

5.193.27 `ssl::context::set_verify_mode`

Set the peer verification mode.

```
void set_verify_mode(
    verify_mode v);

asio::error_code set_verify_mode(
    verify_mode v,
    asio::error_code & ec);
```

5.193.27.1 `ssl::context::set_verify_mode (1 of 2 overloads)`

Set the peer verification mode.

```
void set_verify_mode(
    verify_mode v);
```

This function may be used to configure the peer verification mode used by the context.

Parameters

v A bitmask of peer verification modes. See `ssl::verify_mode` for available values.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_CTX_set_verify`.

5.193.27.2 `ssl::context::set_verify_mode` (2 of 2 overloads)

Set the peer verification mode.

```
asio::error_code set_verify_mode(
    verify_mode v,
    asio::error_code & ec);
```

This function may be used to configure the peer verification mode used by the context.

Parameters

v A bitmask of peer verification modes. See [ssl::verify_mode](#) for available values.

ec Set to indicate what error occurred, if any.

Remarks

Calls `SSL_CTX_set_verify`.

5.193.28 `ssl::context::single_dh_use`

Always create a new key when using `tmp_dh` parameters.

```
static const long single_dh_use = implementation_defined;
```

5.193.29 `ssl::context::use_certificate`

Use a certificate from a memory buffer.

```
void use_certificate(
    const const_buffer & certificate,
    file_format format);

asio::error_code use_certificate(
    const const_buffer & certificate,
    file_format format,
    asio::error_code & ec);
```

5.193.29.1 `ssl::context::use_certificate` (1 of 2 overloads)

Use a certificate from a memory buffer.

```
void use_certificate(
    const const_buffer & certificate,
    file_format format);
```

This function is used to load a certificate into the context from a buffer.

Parameters

certificate The buffer containing the certificate.

format The certificate format (ASN.1 or PEM).

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_CTX_use_certificate or SSL_CTX_use_certificate_ASN1.

5.193.29.2 `ssl::context::use_certificate` (2 of 2 overloads)

Use a certificate from a memory buffer.

```
asio::error_code use_certificate(
    const const_buffer & certificate,
    file_format format,
    asio::error_code & ec);
```

This function is used to load a certificate into the context from a buffer.

Parameters

certificate The buffer containing the certificate.

format The certificate format (ASN.1 or PEM).

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_use_certificate or SSL_CTX_use_certificate_ASN1.

5.193.30 `ssl::context::use_certificate_chain`

Use a certificate chain from a memory buffer.

```
void use_certificate_chain(
    const const_buffer & chain);

asio::error_code use_certificate_chain(
    const const_buffer & chain,
    asio::error_code & ec);
```

5.193.30.1 `ssl::context::use_certificate_chain` (1 of 2 overloads)

Use a certificate chain from a memory buffer.

```
void use_certificate_chain(
    const const_buffer & chain);
```

This function is used to load a certificate chain into the context from a buffer.

Parameters

chain The buffer containing the certificate chain. The certificate chain must use the PEM format.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_CTX_use_certificate` and `SSL_CTX_add_extra_chain_cert`.

5.193.30.2 `ssl::context::use_certificate_chain` (2 of 2 overloads)

Use a certificate chain from a memory buffer.

```
asio::error_code use_certificate_chain(
    const const_buffer & chain,
    asio::error_code & ec);
```

This function is used to load a certificate chain into the context from a buffer.

Parameters

chain The buffer containing the certificate chain. The certificate chain must use the PEM format.

ec Set to indicate what error occurred, if any.

Remarks

Calls `SSL_CTX_use_certificate` and `SSL_CTX_add_extra_chain_cert`.

5.193.31 `ssl::context::use_certificate_chain_file`

Use a certificate chain from a file.

```
void use_certificate_chain_file(
    const std::string & filename);
```

```
asio::error_code use_certificate_chain_file(
    const std::string & filename,
    asio::error_code & ec);
```

5.193.31.1 `ssl::context::use_certificate_chain_file` (1 of 2 overloads)

Use a certificate chain from a file.

```
void use_certificate_chain_file(  
    const std::string & filename);
```

This function is used to load a certificate chain into the context from a file.

Parameters

filename The name of the file containing the certificate. The file must use the PEM format.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_CTX_use_certificate_chain_file`.

5.193.31.2 `ssl::context::use_certificate_chain_file` (2 of 2 overloads)

Use a certificate chain from a file.

```
asio::error_code use_certificate_chain_file(  
    const std::string & filename,  
    asio::error_code & ec);
```

This function is used to load a certificate chain into the context from a file.

Parameters

filename The name of the file containing the certificate. The file must use the PEM format.

ec Set to indicate what error occurred, if any.

Remarks

Calls `SSL_CTX_use_certificate_chain_file`.

5.193.32 `ssl::context::use_certificate_file`

Use a certificate from a file.

```
void use_certificate_file(  
    const std::string & filename,  
    file_format format);  
  
asio::error_code use_certificate_file(  
    const std::string & filename,  
    file_format format,  
    asio::error_code & ec);
```

5.193.32.1 `ssl::context::use_certificate_file` (1 of 2 overloads)

Use a certificate from a file.

```
void use_certificate_file(
    const std::string & filename,
    file_format format);
```

This function is used to load a certificate into the context from a file.

Parameters

filename The name of the file containing the certificate.

format The file format (ASN.1 or PEM).

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_CTX_use_certificate_file`.

5.193.32.2 `ssl::context::use_certificate_file` (2 of 2 overloads)

Use a certificate from a file.

```
asio::error_code use_certificate_file(
    const std::string & filename,
    file_format format,
    asio::error_code & ec);
```

This function is used to load a certificate into the context from a file.

Parameters

filename The name of the file containing the certificate.

format The file format (ASN.1 or PEM).

ec Set to indicate what error occurred, if any.

Remarks

Calls `SSL_CTX_use_certificate_file`.

5.193.33 `ssl::context::use_private_key`

Use a private key from a memory buffer.

```
void use_private_key(
    const const_buffer & private_key,
    file_format format);

asio::error_code use_private_key(
    const const_buffer & private_key,
    file_format format,
    asio::error_code & ec);
```

5.193.33.1 `ssl::context::use_private_key (1 of 2 overloads)`

Use a private key from a memory buffer.

```
void use_private_key(
    const const_buffer & private_key,
    file_format format);
```

This function is used to load a private key into the context from a buffer.

Parameters

private_key The buffer containing the private key.

format The private key format (ASN.1 or PEM).

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_CTX_use_PrivateKey or SSL_CTX_use_PrivateKey_ASN1.

5.193.33.2 `ssl::context::use_private_key (2 of 2 overloads)`

Use a private key from a memory buffer.

```
asio::error_code use_private_key(
    const const_buffer & private_key,
    file_format format,
    asio::error_code & ec);
```

This function is used to load a private key into the context from a buffer.

Parameters

private_key The buffer containing the private key.

format The private key format (ASN.1 or PEM).

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_use_PrivateKey or SSL_CTX_use_PrivateKey_ASN1.

5.193.34 ssl::context::use_private_key_file

Use a private key from a file.

```
void use_private_key_file(
    const std::string & filename,
    file_format format);

asio::error_code use_private_key_file(
    const std::string & filename,
    file_format format,
    asio::error_code & ec);
```

5.193.34.1 ssl::context::use_private_key_file (1 of 2 overloads)

Use a private key from a file.

```
void use_private_key_file(
    const std::string & filename,
    file_format format);
```

This function is used to load a private key into the context from a file.

Parameters

filename The name of the file containing the private key.

format The file format (ASN.1 or PEM).

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_CTX_use_PrivateKey_file.

5.193.34.2 ssl::context::use_private_key_file (2 of 2 overloads)

Use a private key from a file.

```
asio::error_code use_private_key_file(
    const std::string & filename,
    file_format format,
    asio::error_code & ec);
```

This function is used to load a private key into the context from a file.

Parameters

filename The name of the file containing the private key.

format The file format (ASN.1 or PEM).

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_use_PrivateKey_file.

5.193.35 ssl::context::use_rsa_private_key

Use an RSA private key from a memory buffer.

```
void use_rsa_private_key(
    const const_buffer & private_key,
    file_format format);

asio::error_code use_rsa_private_key(
    const const_buffer & private_key,
    file_format format,
    asio::error_code & ec);
```

5.193.35.1 ssl::context::use_rsa_private_key (1 of 2 overloads)

Use an RSA private key from a memory buffer.

```
void use_rsa_private_key(
    const const_buffer & private_key,
    file_format format);
```

This function is used to load an RSA private key into the context from a buffer.

Parameters

private_key The buffer containing the RSA private key.

format The private key format (ASN.1 or PEM).

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_CTX_use_RSAPrivateKey or SSL_CTX_use_RSAPrivateKey_ASN1.

5.193.35.2 `ssl::context::use_rsa_private_key` (2 of 2 overloads)

Use an RSA private key from a memory buffer.

```
asio::error_code use_rsa_private_key(
    const const_buffer & private_key,
    file_format format,
    asio::error_code & ec);
```

This function is used to load an RSA private key into the context from a buffer.

Parameters

private_key The buffer containing the RSA private key.

format The private key format (ASN.1 or PEM).

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_use_RSAPrivateKey or SSL_CTX_use_RSAPrivateKey_ASN1.

5.193.36 `ssl::context::use_rsa_private_key_file`

Use an RSA private key from a file.

```
void use_rsa_private_key_file(
    const std::string & filename,
    file_format format);

asio::error_code use_rsa_private_key_file(
    const std::string & filename,
    file_format format,
    asio::error_code & ec);
```

5.193.36.1 `ssl::context::use_rsa_private_key_file` (1 of 2 overloads)

Use an RSA private key from a file.

```
void use_rsa_private_key_file(
    const std::string & filename,
    file_format format);
```

This function is used to load an RSA private key into the context from a file.

Parameters

filename The name of the file containing the RSA private key.

format The file format (ASN.1 or PEM).

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_CTX_use_RSAPrivateKey_file.

5.193.36.2 ssl::context::use_rsa_private_key_file (2 of 2 overloads)

Use an RSA private key from a file.

```
asio::error_code use_rsa_private_key_file(
    const std::string & filename,
    file_format format,
    asio::error_code & ec);
```

This function is used to load an RSA private key into the context from a file.

Parameters

filename The name of the file containing the RSA private key.

format The file format (ASN.1 or PEM).

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_use_RSAPrivateKey_file.

5.193.37 ssl::context::use_tmp_dh

Use the specified memory buffer to obtain the temporary Diffie-Hellman parameters.

```
void use_tmp_dh(
    const const_buffer & dh);

asio::error_code use_tmp_dh(
    const const_buffer & dh,
    asio::error_code & ec);
```

5.193.37.1 ssl::context::use_tmp_dh (1 of 2 overloads)

Use the specified memory buffer to obtain the temporary Diffie-Hellman parameters.

```
void use_tmp_dh(
    const const_buffer & dh);
```

This function is used to load Diffie-Hellman parameters into the context from a buffer.

Parameters

dh The memory buffer containing the Diffie-Hellman parameters. The buffer must use the PEM format.

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_CTX_set_tmp_dh.

5.193.37.2 ssl::context::use_tmp_dh (2 of 2 overloads)

Use the specified memory buffer to obtain the temporary Diffie-Hellman parameters.

```
asio::error_code use_tmp_dh(
    const const_buffer & dh,
    asio::error_code & ec);
```

This function is used to load Diffie-Hellman parameters into the context from a buffer.

Parameters

dh The memory buffer containing the Diffie-Hellman parameters. The buffer must use the PEM format.

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_set_tmp_dh.

5.193.38 ssl::context::use_tmp_dh_file

Use the specified file to obtain the temporary Diffie-Hellman parameters.

```
void use_tmp_dh_file(
    const std::string & filename);

asio::error_code use_tmp_dh_file(
    const std::string & filename,
    asio::error_code & ec);
```

5.193.38.1 ssl::context::use_tmp_dh_file (1 of 2 overloads)

Use the specified file to obtain the temporary Diffie-Hellman parameters.

```
void use_tmp_dh_file(
    const std::string & filename);
```

This function is used to load Diffie-Hellman parameters into the context from a file.

Parameters

filename The name of the file containing the Diffie-Hellman parameters. The file must use the PEM format.

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_CTX_set_tmp_dh.

5.193.38.2 ssl::context::use_tmp_dh_file (2 of 2 overloads)

Use the specified file to obtain the temporary Diffie-Hellman parameters.

```
asio::error_code use_tmp_dh_file(
    const std::string & filename,
    asio::error_code & ec);
```

This function is used to load Diffie-Hellman parameters into the context from a file.

Parameters

filename The name of the file containing the Diffie-Hellman parameters. The file must use the PEM format.

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_CTX_set_tmp_dh.

5.193.39 ssl::context::~context

Destructor.

```
~context();
```

5.194 ssl::context_base

The `ssl::context_base` class is used as a base for the `basic_context` class template so that we have a common place to define various enums.

```
class context_base
```

Types

Name	Description
file_format	File format types.
method	Different methods supported by a context.
options	Bitmask type for SSL options.
password_purpose	Purpose of PEM password.

Protected Member Functions

Name	Description
~context_base	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
default_workarounds	Implement various bug workarounds.
no_compression	Disable compression. Compression is disabled by default.
no_sslv2	Disable SSL v2.
no_sslv3	Disable SSL v3.
no_tlsv1	Disable TLS v1.
no_tlsv1_1	Disable TLS v1.1.
no_tlsv1_2	Disable TLS v1.2.
single_dh_use	Always create a new key when using tmp_dh parameters.

Requirements

Header: asio/ssl/context_base.hpp

Convenience header: asio/ssl.hpp

5.194.1 ssl::context_base::default_workarounds

Implement various bug workarounds.

```
static const long default_workarounds = implementation_defined;
```

5.194.2 `ssl::context_base::file_format`

File format types.

```
enum file_format
```

Values

asn1 ASN.1 file.

pem PEM file.

5.194.3 `ssl::context_base::method`

Different methods supported by a context.

```
enum method
```

Values

sslv2 Generic SSL version 2.

sslv2_client SSL version 2 client.

sslv2_server SSL version 2 server.

sslv3 Generic SSL version 3.

sslv3_client SSL version 3 client.

sslv3_server SSL version 3 server.

tlsv1 Generic TLS version 1.

tlsv1_client TLS version 1 client.

tlsv1_server TLS version 1 server.

sslv23 Generic SSL/TLS.

sslv23_client SSL/TLS client.

sslv23_server SSL/TLS server.

tlsv11 Generic TLS version 1.1.

tlsv11_client TLS version 1.1 client.

tlsv11_server TLS version 1.1 server.

tlsv12 Generic TLS version 1.2.

tlsv12_client TLS version 1.2 client.

tlsv12_server TLS version 1.2 server.

5.194.4 `ssl::context_base::no_compression`

Disable compression. Compression is disabled by default.

```
static const long no_compression = implementation_defined;
```

5.194.5 `ssl::context_base::no_sslv2`

Disable SSL v2.

```
static const long no_sslv2 = implementation_defined;
```

5.194.6 `ssl::context_base::no_sslv3`

Disable SSL v3.

```
static const long no_sslv3 = implementation_defined;
```

5.194.7 `ssl::context_base::no_tlsv1`

Disable TLS v1.

```
static const long no_tlsv1 = implementation_defined;
```

5.194.8 `ssl::context_base::no_tlsv1_1`

Disable TLS v1.1.

```
static const long no_tlsv1_1 = implementation_defined;
```

5.194.9 `ssl::context_base::no_tlsv1_2`

Disable TLS v1.2.

```
static const long no_tlsv1_2 = implementation_defined;
```

5.194.10 `ssl::context_base::options`

Bitmask type for SSL options.

```
typedef long options;
```

Requirements

Header: `asio/ssl/context_base.hpp`

Convenience header: `asio/ssl.hpp`

5.194.11 `ssl::context_base::password_purpose`

Purpose of PEM password.

```
enum password_purpose
```

Values

for_reading The password is needed for reading/decryption.

for_writing The password is needed for writing/encryption.

5.194.12 `ssl::context_base::single_dh_use`

Always create a new key when using tmp_dh parameters.

```
static const long single_dh_use = implementation_defined;
```

5.194.13 `ssl::context_base::~context_base`

Protected destructor to prevent deletion through this type.

```
~context_base();
```

5.195 `ssl::rfc2818_verification`

Verifies a certificate against a hostname according to the rules described in RFC 2818.

```
class rfc2818_verification
```

Types

Name	Description
<code>result_type</code>	The type of the function object's result.

Member Functions

Name	Description
<code>operator()</code>	Perform certificate verification.
<code>rfc2818_verification</code>	Constructor.

Example

The following example shows how to synchronously open a secure connection to a given host name:

```

using asio::ip::tcp;
namespace ssl = asio::ssl;
typedef ssl::stream<tcp::socket> ssl_socket;

// Create a context that uses the default paths for finding CA certificates.
ssl::context ctx(ssl::context::sslv23);
ctx.set_default_verify_paths();

// Open a socket and connect it to the remote host.
asio::io_service io_service;
ssl_socket sock(io_service, ctx);
tcp::resolver resolver(io_service);
tcp::resolver::query query("host.name", "https");
asio::connect(sock.lowest_layer(), resolver.resolve(query));
sock.lowest_layer().set_option(tcp::no_delay(true));

// Perform SSL handshake and verify the remote host's certificate.
sock.set_verify_mode(ssl::verify_peer);
sock.set_verify_callback(ssl::rfc2818_verification("host.name"));
sock.handshake(ssl_socket::client);

// ... read and write as normal ...

```

Requirements

Header: asio/ssl/rfc2818_verification.hpp

Convenience header: asio/ssl.hpp

5.195.1 ssl::rfc2818_verification::operator()

Perform certificate verification.

```

bool operator()
    bool preverified,
    verify_context & ctx) const;

```

5.195.2 ssl::rfc2818_verification::result_type

The type of the function object's result.

```
typedef bool result_type;
```

Requirements

Header: asio/ssl/rfc2818_verification.hpp

Convenience header: asio/ssl.hpp

5.195.3 ssl::rfc2818_verification::rfc2818_verification

Constructor.

```

rfc2818_verification(
    const std::string & host);

```

5.196 ssl::stream

Provides stream-oriented functionality using SSL.

```
template<
    typename Stream>
class stream :
    public ssl::stream_base,
    noncopyable
```

Types

Name	Description
impl_struct	Structure for use with deprecated impl_type.
handshake_type	Different handshake types.
impl_type	(Deprecated: Use native_handle_type.) The underlying implementation type.
lowest_layer_type	The type of the lowest layer.
native_handle_type	The native handle type of the SSL stream.
next_layer_type	The type of the next layer.

Member Functions

Name	Description
async_handshake	Start an asynchronous SSL handshake.
async_read_some	Start an asynchronous read.
async_shutdown	Asynchronously shut down SSL on the stream.
async_write_some	Start an asynchronous write.
get_io_service	Get the io_service associated with the object.
handshake	Perform SSL handshaking.
impl	(Deprecated: Use native_handle().) Get the underlying implementation in the native type.
lowest_layer	Get a reference to the lowest layer.
native_handle	Get the underlying implementation in the native type.
next_layer	Get a reference to the next layer.
read_some	Read some data from the stream.

Name	Description
set_verify_callback	Set the callback used to verify peer certificates.
set_verify_depth	Set the peer verification depth.
set_verify_mode	Set the peer verification mode.
shutdown	Shut down SSL on the stream.
stream	Construct a stream.
write_some	Write some data to the stream.
~stream	Destructor.

The stream class template provides asynchronous and blocking stream-oriented functionality using SSL.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe. The application must also ensure that all asynchronous operations are performed within the same implicit or explicit strand.

Example

To use the SSL stream template with an `ip::tcp::socket`, you would write:

```
asio::io_service io_service;
asio::ssl::context ctx(asio::ssl::context::sslv23);
asio::ssl::stream<asio::ip::tcp::socket> sock(io_service, ctx);
```

Requirements

Header: `asio/ssl/stream.hpp`

Convenience header: `asio/ssl.hpp`

5.196.1 `ssl::stream::async_handshake`

Start an asynchronous SSL handshake.

```
template<
    typename HandshakeHandler>
void-or-deduced async_handshake(
    handshake_type type,
    HandshakeHandler handler);

template<
    typename ConstBufferSequence,
    typename BufferedHandshakeHandler>
void-or-deduced async_handshake(
    handshake_type type,
    const ConstBufferSequence & buffers,
    BufferedHandshakeHandler handler);
```

5.196.1.1 `ssl::stream::async_handshake` (1 of 2 overloads)

Start an asynchronous SSL handshake.

```
template<
    typename HandshakeHandler>
void-or-deduced async_handshake(
    handshake_type type,
    HandshakeHandler handler);
```

This function is used to asynchronously perform an SSL handshake on the stream. This function call always returns immediately.

Parameters

type The type of handshaking to be performed, i.e. as a client or as a server.

handler The handler to be called when the handshake operation completes. Copies will be made of the handler as required. The equivalent function signature of the handler must be:

```
void handler(
    const asio::error_code& error // Result of operation.
);
```

5.196.1.2 `ssl::stream::async_handshake` (2 of 2 overloads)

Start an asynchronous SSL handshake.

```
template<
    typename ConstBufferSequence,
    typename BufferedHandshakeHandler>
void-or-deduced async_handshake(
    handshake_type type,
    const ConstBufferSequence & buffers,
    BufferedHandshakeHandler handler);
```

This function is used to asynchronously perform an SSL handshake on the stream. This function call always returns immediately.

Parameters

type The type of handshaking to be performed, i.e. as a client or as a server.

buffers The buffered data to be reused for the handshake. Although the buffers object may be copied as necessary, ownership of the underlying buffers is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the handshake operation completes. Copies will be made of the handler as required. The equivalent function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred // Amount of buffers used in handshake.
);
```

5.196.2 ssl::stream::async_read_some

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

This function is used to asynchronously read one or more bytes of data from the stream. The function call always returns immediately.

Parameters

buffers The buffers into which the data will be read. Although the buffers object may be copied as necessary, ownership of the underlying buffers is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The equivalent function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes read.
);
```

Remarks

The `async_read_some` operation may not read all of the requested number of bytes. Consider using the `async_read` function if you need to ensure that the requested amount of data is read before the asynchronous operation completes.

5.196.3 ssl::stream::async_shutdown

Asynchronously shut down SSL on the stream.

```
template<
    typename ShutdownHandler>
void-or-deduced async_shutdown(
    ShutdownHandler handler);
```

This function is used to asynchronously shut down SSL on the stream. This function call always returns immediately.

Parameters

handler The handler to be called when the handshake operation completes. Copies will be made of the handler as required. The equivalent function signature of the handler must be:

```
void handler(
    const asio::error_code& error // Result of operation.
);
```

5.196.4 ssl::stream::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some(
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

This function is used to asynchronously write one or more bytes of data to the stream. The function call always returns immediately.

Parameters

buffers The data to be written to the stream. Although the buffers object may be copied as necessary, ownership of the underlying buffers is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The equivalent function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes written.
);
```

Remarks

The `async_write_some` operation may not transmit all of the data to the peer. Consider using the `async_write` function if you need to ensure that all data is written before the blocking operation completes.

5.196.5 ssl::stream::get_io_service

Get the `io_service` associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the `io_service` object that the stream uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_service` object that stream will use to dispatch handlers. Ownership is not transferred to the caller.

5.196.6 ssl::stream::handshake

Perform SSL handshaking.

```
void handshake(
    handshake_type type);

asio::error_code handshake(
    handshake_type type,
    asio::error_code & ec);
```

```

template<
    typename ConstBufferSequence>
void handshake(
    handshake_type type,
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
asio::error_code handshake(
    handshake_type type,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);

```

5.196.6.1 `ssl::stream::handshake` (1 of 4 overloads)

Perform SSL handshaking.

```
void handshake(
    handshake_type type);
```

This function is used to perform SSL handshaking on the stream. The function call will block until handshaking is complete or an error occurs.

Parameters

type The type of handshaking to be performed, i.e. as a client or as a server.

Exceptions

asio::system_error Thrown on failure.

5.196.6.2 `ssl::stream::handshake` (2 of 4 overloads)

Perform SSL handshaking.

```
asio::error_code handshake(
    handshake_type type,
    asio::error_code & ec);
```

This function is used to perform SSL handshaking on the stream. The function call will block until handshaking is complete or an error occurs.

Parameters

type The type of handshaking to be performed, i.e. as a client or as a server.

ec Set to indicate what error occurred, if any.

5.196.6.3 `ssl::stream::handshake` (3 of 4 overloads)

Perform SSL handshaking.

```
template<
    typename ConstBufferSequence>
void handshake(
    handshake_type type,
    const ConstBufferSequence & buffers);
```

This function is used to perform SSL handshaking on the stream. The function call will block until handshaking is complete or an error occurs.

Parameters

type The type of handshaking to be performed, i.e. as a client or as a server.

buffers The buffered data to be reused for the handshake.

Exceptions

`asio::system_error` Thrown on failure.

5.196.6.4 `ssl::stream::handshake` (4 of 4 overloads)

Perform SSL handshaking.

```
template<
    typename ConstBufferSequence>
asio::error_code handshake(
    handshake_type type,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to perform SSL handshaking on the stream. The function call will block until handshaking is complete or an error occurs.

Parameters

type The type of handshaking to be performed, i.e. as a client or as a server.

buffers The buffered data to be reused for the handshake.

ec Set to indicate what error occurred, if any.

5.196.7 `ssl::stream::handshake_type`

Different handshake types.

```
enum handshake_type
```

Values

client Perform handshaking as a client.

server Perform handshaking as a server.

5.196.8 `ssl::stream::impl`

(Deprecated: Use `native_handle()`.) Get the underlying implementation in the native type.

```
impl_type impl();
```

This function may be used to obtain the underlying implementation of the context. This is intended to allow access to stream functionality that is not otherwise provided.

5.196.9 `ssl::stream::impl_type`

(Deprecated: Use `native_handle_type`.) The underlying implementation type.

```
typedef impl_struct * impl_type;
```

Requirements

Header: `asio/ssl/stream.hpp`

Convenience header: `asio/ssl.hpp`

5.196.10 `ssl::stream::lowest_layer`

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();  
  
const lowest_layer_type & lowest_layer() const;
```

5.196.10.1 `ssl::stream::lowest_layer (1 of 2 overloads)`

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of stream layers.

Return Value

A reference to the lowest layer in the stack of stream layers. Ownership is not transferred to the caller.

5.196.10.2 `ssl::stream::lowest_layer (2 of 2 overloads)`

Get a reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a reference to the lowest layer in a stack of stream layers.

Return Value

A reference to the lowest layer in the stack of stream layers. Ownership is not transferred to the caller.

5.196.11 `ssl::stream::lowest_layer_type`

The type of the lowest layer.

```
typedef next_layer_type::lowest_layer_type lowest_layer_type;
```

Requirements

Header: asio/ssl/stream.hpp

Convenience header: asio/ssl.hpp

5.196.12 `ssl::stream::native_handle`

Get the underlying implementation in the native type.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying implementation of the context. This is intended to allow access to context functionality that is not otherwise provided.

Example

The `native_handle()` function returns a pointer of type `SSL*` that is suitable for passing to functions such as `SSL_get_verify_result` and `SSL_get_peer_certificate`:

```
asio::ssl::stream<asio::ip::tcp::socket> sock(io_service, ctx);

// ... establish connection and perform handshake ...

if (X509* cert = SSL_get_peer_certificate(sock.native_handle()))
{
    if (SSL_get_verify_result(sock.native_handle()) == X509_V_OK)
    {
        // ...
    }
}
```

5.196.13 `ssl::stream::native_handle_type`

The native handle type of the SSL stream.

```
typedef SSL * native_handle_type;
```

Requirements

Header: asio/ssl/stream.hpp

Convenience header: asio/ssl.hpp

5.196.14 `ssl::stream::next_layer`

Get a reference to the next layer.

```
const next_layer_type & next_layer() const;
```

```
next_layer_type & next_layer();
```

5.196.14.1 `ssl::stream::next_layer` (1 of 2 overloads)

Get a reference to the next layer.

```
const next_layer_type & next_layer() const;
```

This function returns a reference to the next layer in a stack of stream layers.

Return Value

A reference to the next layer in the stack of stream layers. Ownership is not transferred to the caller.

5.196.14.2 `ssl::stream::next_layer` (2 of 2 overloads)

Get a reference to the next layer.

```
next_layer_type & next_layer();
```

This function returns a reference to the next layer in a stack of stream layers.

Return Value

A reference to the next layer in the stack of stream layers. Ownership is not transferred to the caller.

5.196.15 `ssl::stream::next_layer_type`

The type of the next layer.

```
typedef remove_reference< Stream >::type next_layer_type;
```

Requirements

Header: asio/ssl/stream.hpp

Convenience header: asio/ssl.hpp

5.196.16 `ssl::stream::read_some`

Read some data from the stream.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);

template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.196.16.1 `ssl::stream::read_some` (1 of 2 overloads)

Read some data from the stream.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);
```

This function is used to read data from the stream. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

buffers The buffers into which the data will be read.

Return Value

The number of bytes read.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The `read_some` operation may not read all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

5.196.16.2 `ssl::stream::read_some` (2 of 2 overloads)

Read some data from the stream.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to read data from the stream. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

buffers The buffers into which the data will be read.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes read. Returns 0 if an error occurred.

Remarks

The `read_some` operation may not read all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

5.196.17 `ssl::stream::set_verify_callback`

Set the callback used to verify peer certificates.

```
template<
    typename VerifyCallback>
void set_verify_callback(
    VerifyCallback callback);

template<
    typename VerifyCallback>
asio::error_code set_verify_callback(
    VerifyCallback callback,
    asio::error_code & ec);
```

5.196.17.1 `ssl::stream::set_verify_callback (1 of 2 overloads)`

Set the callback used to verify peer certificates.

```
template<
    typename VerifyCallback>
void set_verify_callback(
    VerifyCallback callback);
```

This function is used to specify a callback function that will be called by the implementation when it needs to verify a peer certificate.

Parameters

callback The function object to be used for verifying a certificate. The function signature of the handler must be:

```
bool verify_callback(
    bool preverified, // True if the certificate passed pre-verification.
    verify_context& ctx // The peer certificate and other context.
);
```

The return value of the callback is true if the certificate has passed verification, false otherwise.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

Calls `SSL_set_verify`.

5.196.17.2 `ssl::stream::set_verify_callback` (2 of 2 overloads)

Set the callback used to verify peer certificates.

```
template<
    typename VerifyCallback>
asio::error_code set_verify_callback(
    VerifyCallback callback,
    asio::error_code & ec);
```

This function is used to specify a callback function that will be called by the implementation when it needs to verify a peer certificate.

Parameters

callback The function object to be used for verifying a certificate. The function signature of the handler must be:

```
bool verify_callback(
    bool preverified, // True if the certificate passed pre-verification.
    verify_context& ctx // The peer certificate and other context.
);
```

The return value of the callback is true if the certificate has passed verification, false otherwise.

ec Set to indicate what error occurred, if any.

Remarks

Calls `SSL_set_verify`.

5.196.18 `ssl::stream::set_verify_depth`

Set the peer verification depth.

```
void set_verify_depth(
    int depth);

asio::error_code set_verify_depth(
    int depth,
    asio::error_code & ec);
```

5.196.18.1 `ssl::stream::set_verify_depth` (1 of 2 overloads)

Set the peer verification depth.

```
void set_verify_depth(
    int depth);
```

This function may be used to configure the maximum verification depth allowed by the stream.

Parameters

depth Maximum depth for the certificate chain verification that shall be allowed.

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_set_verify_depth.

5.196.18.2 ssl::stream::set_verify_depth (2 of 2 overloads)

Set the peer verification depth.

```
asio::error_code set_verify_depth(
    int depth,
    asio::error_code & ec);
```

This function may be used to configure the maximum verification depth allowed by the stream.

Parameters

depth Maximum depth for the certificate chain verification that shall be allowed.

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_set_verify_depth.

5.196.19 ssl::stream::set_verify_mode

Set the peer verification mode.

```
void set_verify_mode(
    verify_mode v);

asio::error_code set_verify_mode(
    verify_mode v,
    asio::error_code & ec);
```

5.196.19.1 ssl::stream::set_verify_mode (1 of 2 overloads)

Set the peer verification mode.

```
void set_verify_mode(
    verify_mode v);
```

This function may be used to configure the peer verification mode used by the stream. The new mode will override the mode inherited from the context.

Parameters

v A bitmask of peer verification modes. See [ssl::verify_mode](#) for available values.

Exceptions

asio::system_error Thrown on failure.

Remarks

Calls SSL_set_verify.

5.196.19.2 **ssl::stream::set_verify_mode (2 of 2 overloads)**

Set the peer verification mode.

```
asio::error_code set_verify_mode(
    verify_mode v,
    asio::error_code & ec);
```

This function may be used to configure the peer verification mode used by the stream. The new mode will override the mode inherited from the context.

Parameters

v A bitmask of peer verification modes. See [ssl::verify_mode](#) for available values.

ec Set to indicate what error occurred, if any.

Remarks

Calls SSL_set_verify.

5.196.20 **ssl::stream::shutdown**

Shut down SSL on the stream.

```
void shutdown();

asio::error_code shutdown(
    asio::error_code & ec);
```

5.196.20.1 **ssl::stream::shutdown (1 of 2 overloads)**

Shut down SSL on the stream.

```
void shutdown();
```

This function is used to shut down SSL on the stream. The function call will block until SSL has been shut down or an error occurs.

Exceptions

asio::system_error Thrown on failure.

5.196.20.2 `ssl::stream::shutdown` (2 of 2 overloads)

Shut down SSL on the stream.

```
asio::error_code shutdown(
    asio::error_code & ec);
```

This function is used to shut down SSL on the stream. The function call will block until SSL has been shut down or an error occurs.

Parameters

ec Set to indicate what error occurred, if any.

5.196.21 `ssl::stream::stream`

Construct a stream.

```
template<
    typename Arg>
stream(
    Arg & arg,
    context & ctx);
```

This constructor creates a stream and initialises the underlying stream object.

Parameters

arg The argument to be passed to initialise the underlying stream.

ctx The SSL context to be used for the stream.

5.196.22 `ssl::stream::write_some`

Write some data to the stream.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.196.22.1 `ssl::stream::write_some` (1 of 2 overloads)

Write some data to the stream.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

This function is used to write data on the stream. The function call will block until one or more bytes of data has been written successfully, or until an error occurs.

Parameters

buffers The data to be written.

Return Value

The number of bytes written.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

The write_some operation may not transmit all of the data to the peer. Consider using the `write` function if you need to ensure that all data is written before the blocking operation completes.

5.196.22.2 `ssl::stream::write_some` (2 of 2 overloads)

Write some data to the stream.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to write data on the stream. The function call will block until one or more bytes of data has been written successfully, or until an error occurs.

Parameters

buffers The data to be written to the stream.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes written. Returns 0 if an error occurred.

Remarks

The write_some operation may not transmit all of the data to the peer. Consider using the `write` function if you need to ensure that all data is written before the blocking operation completes.

5.196.23 `ssl::stream::~stream`

Destructor.

```
~stream();
```

5.197 ssl::stream::impl_struct

Structure for use with deprecated impl_type.

```
struct impl_struct
```

Data Members

Name	Description
ssl	

Requirements

Header: asio/ssl/stream.hpp

Convenience header: asio/ssl.hpp

5.197.1 ssl::stream::impl_struct::ssl

```
SSL * ssl;
```

5.198 ssl::stream_base

The `ssl::stream_base` class is used as a base for the `ssl::stream` class template so that we have a common place to define various enums.

```
class stream_base
```

Types

Name	Description
handshake_type	Different handshake types.

Protected Member Functions

Name	Description
<code>~stream_base</code>	Protected destructor to prevent deletion through this type.

Requirements

Header: asio/ssl/stream_base.hpp

Convenience header: asio/ssl.hpp

5.198.1 `ssl::stream_base::handshake_type`

Different handshake types.

```
enum handshake_type
```

Values

client Perform handshaking as a client.

server Perform handshaking as a server.

5.198.2 `ssl::stream_base::~stream_base`

Protected destructor to prevent deletion through this type.

```
~stream_base();
```

5.199 `ssl::verify_client_once`

Do not request client certificate on renegotiation. Ignored unless `ssl::verify_peer` is set.

```
const int verify_client_once = implementation_defined;
```

Requirements

Header: `asio/ssl/verify_mode.hpp`

Convenience header: `asio/ssl.hpp`

5.200 `ssl::verify_context`

A simple wrapper around the X509_STORE_CTX type, used during verification of a peer certificate.

```
class verify_context :  
    noncopyable
```

Types

Name	Description
<code>native_handle_type</code>	The native handle type of the verification context.

Member Functions

Name	Description
<code>native_handle</code>	Get the underlying implementation in the native type.

Name	Description
verify_context	Constructor.

Remarks

The `ssl::verify_context` does not own the underlying X509_STORE_CTX object.

Requirements

Header: asio/ssl/verify_context.hpp

Convenience header: asio/ssl.hpp

5.200.1 `ssl::verify_context::native_handle`

Get the underlying implementation in the native type.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying implementation of the context. This is intended to allow access to context functionality that is not otherwise provided.

5.200.2 `ssl::verify_context::native_handle_type`

The native handle type of the verification context.

```
typedef X509_STORE_CTX * native_handle_type;
```

Requirements

Header: asio/ssl/verify_context.hpp

Convenience header: asio/ssl.hpp

5.200.3 `ssl::verify_context::verify_context`

Constructor.

```
verify_context(
    native_handle_type handle);
```

5.201 `ssl::verify_fail_if_no_peer_cert`

Fail verification if the peer has no certificate. Ignored unless `ssl::verify_peer` is set.

```
const int verify_fail_if_no_peer_cert = implementation_defined;
```

Requirements

Header: asio/ssl/verify_mode.hpp

Convenience header: asio/ssl.hpp

5.202 `ssl::verify_mode`

Bitmask type for peer verification.

```
typedef int verify_mode;
```

Possible values are:

- `ssl::verify_none`
- `ssl::verify_peer`
- `ssl::verify_fail_if_no_peer_cert`
- `ssl::verify_client_once`

Requirements

Header: `asio/ssl/verify_mode.hpp`

Convenience header: `asio/ssl.hpp`

5.203 `ssl::verify_none`

No verification.

```
const int verify_none = implementation_defined;
```

Requirements

Header: `asio/ssl/verify_mode.hpp`

Convenience header: `asio/ssl.hpp`

5.204 `ssl::verify_peer`

Verify the peer.

```
const int verify_peer = implementation_defined;
```

Requirements

Header: `asio/ssl/verify_mode.hpp`

Convenience header: `asio/ssl.hpp`

5.205 `steady_timer`

Typedef for a timer based on the steady clock.

```
typedef basic_waitable_timer< chrono::steady_clock > steady_timer;
```

Types

Name	Description
clock_type	The clock type.
duration	The duration type of the clock.
implementation_type	The underlying implementation type of I/O object.
service_type	The type of the service that will be used to provide I/O operations.
time_point	The time point type of the clock.
traits_type	The wait traits type.

Member Functions

Name	Description
async_wait	Start an asynchronous wait on the timer.
basic_waitable_timer	Constructor. Constructor to set a particular expiry time as an absolute time. Constructor to set a particular expiry time relative to now.
cancel	Cancel any asynchronous operations that are waiting on the timer.
cancel_one	Cancels one asynchronous operation that is waiting on the timer.
expires_at	Get the timer's expiry time as an absolute time. Set the timer's expiry time as an absolute time.
expires_from_now	Get the timer's expiry time relative to now. Set the timer's expiry time relative to now.
get_io_service	Get the io_service associated with the object.
wait	Perform a blocking wait on the timer.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_waitable_timer` class template provides the ability to perform a blocking or asynchronous wait for a timer to expire.

A waitable timer is always in one of two states: "expired" or "not expired". If the `wait()` or `async_wait()` function is called on an expired timer, the wait operation will complete immediately.

Most applications will use one of the `steady_timer`, `system_timer` or `high_resolution_timer` typedefs.

Remarks

This waitable timer functionality is for use with the C++11 standard library's `<chrono>` facility, or with the Boost.Chrono library.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Examples

Performing a blocking wait (C++11):

```
// Construct a timer without setting an expiry time.
asio::steady_timer timer(io_service);

// Set an expiry time relative to now.
timer.expires_from_now(std::chrono::seconds(5));

// Wait for the timer to expire.
timer.wait();
```

Performing an asynchronous wait (C++11):

```
void handler(const asio::error_code& error)
{
    if (!error)
    {
        // Timer expired.
    }
}

...

// Construct a timer with an absolute expiry time.
asio::steady_timer timer(io_service,
    std::chrono::steady_clock::now() + std::chrono::seconds(60));

// Start an asynchronous wait.
timer.async_wait(handler);
```

Changing an active waitable timer's expiry time

Changing the expiry time of a timer while there are pending asynchronous waits causes those wait operations to be cancelled. To ensure that the action associated with the timer is performed only once, use something like this:

```
void on_some_event()
{
    if (my_timer.expires_from_now(seconds(5)) > 0)
    {
        // We managed to cancel the timer. Start new asynchronous wait.
        my_timer.async_wait(on_timeout);
    }
    else
    {
        // Too late, timer has already expired!
    }
}

void on_timeout(const asio::error_code& e)
{
    if (e != asio::error::operation_aborted)
    {
        // Timer was not cancelled, take necessary action.
    }
}
```

- The `asio::basic_waitable_timer::expires_from_now()` function cancels any pending asynchronous waits, and returns the number of asynchronous waits that were cancelled. If it returns 0 then you were too late and the wait handler has already been executed, or will soon be executed. If it returns 1 then the wait handler was successfully cancelled.
- If a wait handler is cancelled, the `error_code` passed to it contains the value `asio::error::operation_aborted`.

This typedef uses the C++11 `<chrono>` standard library facility, if available. Otherwise, it may use the Boost.Chrono library. To explicitly utilise Boost.Chrono, use the `basic_waitable_timer` template directly:

```
typedef basic_waitable_timer<boost::chrono::steady_clock> timer;
```

Requirements

Header: `asio/steady_timer.hpp`

Convenience header: None

5.206 strand

(Deprecated: Use `io_service::strand`.) Typedef for backwards compatibility.

```
typedef asio::io_service::strand strand;
```

Member Functions

Name	Description
<code>dispatch</code>	Request the strand to invoke the given handler.

Name	Description
get_io_service	Get the io_service associated with the strand.
post	Request the strand to invoke the given handler and return immediately.
running_in_this_thread	Determine whether the strand is running in the current thread.
strand	Constructor.
wrap	Create a new handler that automatically dispatches the wrapped handler on the strand.
~strand	Destructor.

The `io_service::strand` class provides the ability to post and dispatch handlers with the guarantee that none of those handlers will execute concurrently.

Order of handler invocation

Given:

- a strand object `s`
- an object `a` meeting completion handler requirements
- an object `a1` which is an arbitrary copy of `a` made by the implementation
- an object `b` meeting completion handler requirements
- an object `b1` which is an arbitrary copy of `b` made by the implementation

if any of the following conditions are true:

- `s.post(a)` happens-before `s.post(b)`
- `s.post(a)` happens-before `s.dispatch(b)`, where the latter is performed outside the strand
- `s.dispatch(a)` happens-before `s.post(b)`, where the former is performed outside the strand
- `s.dispatch(a)` happens-before `s.dispatch(b)`, where both are performed outside the strand

then `asio_handler_invoke(a1, &a1)` happens-before `asio_handler_invoke(b1, &b1)`.

Note that in the following case:

```
async_op_1(..., s.wrap(a));
async_op_2(..., s.wrap(b));
```

the completion of the first async operation will perform `s.dispatch(a)`, and the second will perform `s.dispatch(b)`, but the order in which those are performed is unspecified. That is, you cannot state whether one happens-before the other. Therefore none of the above conditions are met and no ordering guarantee is made.

Remarks

The implementation makes no guarantee that handlers posted or dispatched through different `strand` objects will be invoked concurrently.

Thread Safety

Distinct objects: Safe.

Shared objects: Safe.

Requirements

Header: asio/stream_socket.hpp

Convenience header: asio.hpp

5.207 stream_socket_service

Default service implementation for a stream socket.

```
template<
    typename Protocol>
class stream_socket_service :
    public io_service::service
```

Types

Name	Description
endpoint_type	The endpoint type.
implementation_type	The type of a stream socket implementation.
native_handle_type	The native socket type.
native_type	(Deprecated: Use native_handle_type.) The native socket type.
protocol_type	The protocol type.

Member Functions

Name	Description
assign	Assign an existing native socket to a stream socket.
async_connect	Start an asynchronous connect.
async_receive	Start an asynchronous receive.
async_send	Start an asynchronous send.
at_mark	Determine whether the socket is at the out-of-band data mark.
available	Determine the number of bytes available for reading.

Name	Description
bind	Bind the stream socket to the specified local endpoint.
cancel	Cancel all asynchronous operations associated with the socket.
close	Close a stream socket implementation.
connect	Connect the stream socket to the specified endpoint.
construct	Construct a new stream socket implementation.
converting_move_construct	Move-construct a new stream socket implementation from another protocol type.
destroy	Destroy a stream socket implementation.
get_io_service	Get the io_service object that owns the service.
get_option	Get a socket option.
io_control	Perform an IO control command on the socket.
is_open	Determine whether the socket is open.
local_endpoint	Get the local endpoint.
move_assign	Move-assign from another stream socket implementation.
move_construct	Move-construct a new stream socket implementation.
native	(Deprecated: Use native_handle().) Get the native socket implementation.
native_handle	Get the native socket implementation.
native_non_blocking	Gets the non-blocking mode of the native socket implementation. Sets the non-blocking mode of the native socket implementation.
non_blocking	Gets the non-blocking mode of the socket. Sets the non-blocking mode of the socket.
open	Open a stream socket.
receive	Receive some data from the peer.
remote_endpoint	Get the remote endpoint.
send	Send the given data to the peer.
set_option	Set a socket option.
shutdown	Disable sends or receives on the socket.

Name	Description
stream_socket_service	Construct a new stream socket service for the specified io_service.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/stream_socket_service.hpp

Convenience header: asio.hpp

5.207.1 stream_socket_service::assign

Assign an existing native socket to a stream socket.

```
asio::error_code assign(
    implementation_type & impl,
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.207.2 stream_socket_service::async_connect

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
void-or-deduced async_connect(
    implementation_type & impl,
    const endpoint_type & peer_endpoint,
    ConnectHandler handler);
```

5.207.3 stream_socket_service::async_receive

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_receive(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    ReadHandler handler);
```

5.207.4 stream_socket_service::async_send

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_send(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler handler);
```

5.207.5 stream_socket_service::at_mark

Determine whether the socket is at the out-of-band data mark.

```
bool at_mark(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.207.6 stream_socket_service::available

Determine the number of bytes available for reading.

```
std::size_t available(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.207.7 stream_socket_service::bind

Bind the stream socket to the specified local endpoint.

```
asio::error_code bind(
    implementation_type & impl,
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.207.8 stream_socket_service::cancel

Cancel all asynchronous operations associated with the socket.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.207.9 stream_socket_service::close

Close a stream socket implementation.

```
asio::error_code close(
    implementation_type & impl,
    asio::error_code & ec);
```

5.207.10 stream_socket_service::connect

Connect the stream socket to the specified endpoint.

```
asio::error_code connect(
    implementation_type & impl,
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.207.11 stream_socket_service::construct

Construct a new stream socket implementation.

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

5.207.12 stream_socket_service::converting_move_construct

Move-construct a new stream socket implementation from another protocol type.

```
template<
    typename Protocol1>
void converting_move_construct(
    implementation_type & impl,
    typename stream_socket_service< Protocol1 >::implementation_type & other_impl,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.207.13 stream_socket_service::destroy

Destroy a stream socket implementation.

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

5.207.14 stream_socket_service::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/stream_socket_service.hpp

Convenience header: asio.hpp

5.207.15 stream_socket_service::get_io_service

Inherited from io_service.

Get the **io_service** object that owns the service.

```
asio::io_service & get_io_service();
```

5.207.16 stream_socket_service::get_option

Get a socket option.

```
template<
    typename GettableSocketOption>
asio::error_code get_option(
    const implementation_type & impl,
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.207.17 stream_socket_service::id

The unique service identifier.

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

5.207.18 stream_socket_service::implementation_type

The type of a stream socket implementation.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asio/stream_socket_service.hpp

Convenience header: asio.hpp

5.207.19 stream_socket_service::io_control

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
asio::error_code io_control(
    implementation_type & impl,
    IoControlCommand & command,
    asio::error_code & ec);
```

5.207.20 stream_socket_service::is_open

Determine whether the socket is open.

```
bool is_open(
    const implementation_type & impl) const;
```

5.207.21 stream_socket_service::local_endpoint

Get the local endpoint.

```
endpoint_type local_endpoint(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.207.22 stream_socket_service::move_assign

Move-assign from another stream socket implementation.

```
void move_assign(
    implementation_type & impl,
    stream_socket_service & other_service,
    implementation_type & other_impl);
```

5.207.23 stream_socket_service::move_construct

Move-construct a new stream socket implementation.

```
void move_construct(
    implementation_type & impl,
    implementation_type & other_impl);
```

5.207.24 stream_socket_service::native

(Deprecated: Use native_handle().) Get the native socket implementation.

```
native_type native(
    implementation_type & impl);
```

5.207.25 stream_socket_service::native_handle

Get the native socket implementation.

```
native_handle_type native_handle(
    implementation_type & impl);
```

5.207.26 stream_socket_service::native_handle_type

The native socket type.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/stream_socket_service.hpp

Convenience header: asio.hpp

5.207.27 stream_socket_service::native_non_blocking

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the native socket implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.207.27.1 stream_socket_service::native_non_blocking (1 of 2 overloads)

Gets the non-blocking mode of the native socket implementation.

```
bool native_non_blocking(
    const implementation_type & impl) const;
```

5.207.27.2 stream_socket_service::native_non_blocking (2 of 2 overloads)

Sets the non-blocking mode of the native socket implementation.

```
asio::error_code native_non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.207.28 stream_socket_service::native_type

(Deprecated: Use native_handle_type.) The native socket type.

```
typedef implementation_defined native_type;
```

Requirements

Header: asio/stream_socket_service.hpp

Convenience header: asio.hpp

5.207.29 stream_socket_service::non_blocking

Gets the non-blocking mode of the socket.

```
bool non_blocking(
    const implementation_type & impl) const;
```

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.207.29.1 stream_socket_service::non_blocking (1 of 2 overloads)

Gets the non-blocking mode of the socket.

```
bool non_blocking(
    const implementation_type & impl) const;
```

5.207.29.2 stream_socket_service::non_blocking (2 of 2 overloads)

Sets the non-blocking mode of the socket.

```
asio::error_code non_blocking(
    implementation_type & impl,
    bool mode,
    asio::error_code & ec);
```

5.207.30 stream_socket_service::open

Open a stream socket.

```
asio::error_code open(
    implementation_type & impl,
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.207.31 stream_socket_service::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/stream_socket_service.hpp

Convenience header: asio.hpp

5.207.32 stream_socket_service::receive

Receive some data from the peer.

```
template<
    typename MutableBufferSequence>
std::size_t receive(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.207.33 stream_socket_service::remote_endpoint

Get the remote endpoint.

```
endpoint_type remote_endpoint(
    const implementation_type & impl,
    asio::error_code & ec) const;
```

5.207.34 stream_socket_service::send

Send the given data to the peer.

```
template<
    typename ConstBufferSequence>
std::size_t send(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    asio::error_code & ec);
```

5.207.35 stream_socket_service::set_option

Set a socket option.

```
template<
    typename SettableSocketOption>
asio::error_code set_option(
    implementation_type & impl,
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.207.36 stream_socket_service::shutdown

Disable sends or receives on the socket.

```
asio::error_code shutdown(
    implementation_type & impl,
    socket_base::shutdown_type what,
    asio::error_code & ec);
```

5.207.37 stream_socket_service::stream_socket_service

Construct a new stream socket service for the specified [io_service](#).

```
stream_socket_service(
    asio::io_service & io_service);
```

5.208 streambuf

TypeDef for the typical usage of [basic_streambuf](#).

```
typedef basic_streambuf streambuf;
```

Types

Name	Description
const_buffers_type	The type used to represent the input sequence as a list of buffers.
mutable_buffers_type	The type used to represent the output sequence as a list of buffers.

Member Functions

Name	Description
<code>basic_streambuf</code>	Construct a <code>basic_streambuf</code> object.
<code>commit</code>	Move characters from the output sequence to the input sequence.
<code>consume</code>	Remove characters from the input sequence.
<code>data</code>	Get a list of buffers that represents the input sequence.
<code>max_size</code>	Get the maximum size of the <code>basic_streambuf</code> .
<code>prepare</code>	Get a list of buffers that represents the output sequence, with the given size.
<code>size</code>	Get the size of the input sequence.

Protected Member Functions

Name	Description
<code>overflow</code>	Override <code>std::streambuf</code> behaviour.
<code>reserve</code>	
<code>underflow</code>	Override <code>std::streambuf</code> behaviour.

The `basic_streambuf` class is derived from `std::streambuf` to associate the `streambuf`'s input and output sequences with one or more character arrays. These character arrays are internal to the `basic_streambuf` object, but direct access to the array elements is provided to permit them to be used efficiently with I/O operations. Characters written to the output sequence of a `basic_streambuf` object are appended to the input sequence of the same object.

The `basic_streambuf` class's public interface is intended to permit the following implementation strategies:

- A single contiguous character array, which is reallocated as necessary to accommodate changes in the size of the character sequence. This is the implementation approach currently used in Asio.
- 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_streambuf` 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 `basic_streambuf` object, the following invariant holds:

```
size() <= max_size()
```

Any member function that would, if successful, cause the invariant to be violated shall throw an exception of class `std::length_error`.

The constructor for `basic_streambuf` 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 `basic_streambuf` object.

Examples

Writing directly from an `streambuf` to a socket:

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

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

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

Reading from a socket directly into a `streambuf`:

```
asio::streambuf b;

// reserve 512 bytes in output sequence
asio::streambuf::mutable_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;
```

Requirements

Header: `asio/streambuf.hpp`

Convenience header: `asio.hpp`

5.209 system_category

Returns the error category used for the system errors produced by `asio`.

```
const error_category & system_category();
```

Requirements

Header: `asio/error_code.hpp`

Convenience header: `asio.hpp`

5.210 system_error

The `system_error` class is used to represent system conditions that prevent the library from operating correctly.

```
class system_error :
    public std::exception
```

Member Functions

Name	Description
code	Get the error code associated with the exception.
operator=	Assignment operator.
system_error	Construct with an error code. Construct with an error code and context. Copy constructor.
what	Get a string representation of the exception.
~system_error	Destructor.

Requirements

Header: asio/system_error.hpp

Convenience header: asio.hpp

5.210.1 system_error::code

Get the error code associated with the exception.

```
error_code code() const;
```

5.210.2 system_error::operator=

Assignment operator.

```
system_error & operator=(  
    const system_error & e);
```

5.210.3 system_error::system_error

Construct with an error code.

```
system_error(  
    const error_code & ec);
```

Construct with an error code and context.

```
system_error(  
    const error_code & ec,  
    const std::string & context);
```

Copy constructor.

```
system_error(  
    const system_error & other);
```

5.210.3.1 system_error::system_error (1 of 3 overloads)

Construct with an error code.

```
system_error(
    const error_code & ec);
```

5.210.3.2 system_error::system_error (2 of 3 overloads)

Construct with an error code and context.

```
system_error(
    const error_code & ec,
    const std::string & context);
```

5.210.3.3 system_error::system_error (3 of 3 overloads)

Copy constructor.

```
system_error(
    const system_error & other);
```

5.210.4 system_error::what

Get a string representation of the exception.

```
virtual const char * what() const;
```

5.210.5 system_error::~system_error

Destructor.

```
virtual ~system_error();
```

5.211 system_timer

Typedef for a timer based on the system clock.

```
typedef basic_waitable_timer< chrono::system_clock > system_timer;
```

Types

Name	Description
clock_type	The clock type.
duration	The duration type of the clock.
implementation_type	The underlying implementation type of I/O object.
service_type	The type of the service that will be used to provide I/O operations.
time_point	The time point type of the clock. 1277
traits_type	The wait traits type.

Member Functions

Name	Description
async_wait	Start an asynchronous wait on the timer.
basic_writable_timer	Constructor. Constructor to set a particular expiry time as an absolute time. Constructor to set a particular expiry time relative to now.
cancel	Cancel any asynchronous operations that are waiting on the timer.
cancel_one	Cancels one asynchronous operation that is waiting on the timer.
expires_at	Get the timer's expiry time as an absolute time. Set the timer's expiry time as an absolute time.
expires_from_now	Get the timer's expiry time relative to now. Set the timer's expiry time relative to now.
get_io_service	Get the io_service associated with the object.
wait	Perform a blocking wait on the timer.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `basic_writable_timer` class template provides the ability to perform a blocking or asynchronous wait for a timer to expire.

A writable timer is always in one of two states: "expired" or "not expired". If the `wait()` or `async_wait()` function is called on an expired timer, the wait operation will complete immediately.

Most applications will use one of the `steady_timer`, `system_timer` or `high_resolution_timer` typedefs.

Remarks

This waitable timer functionality is for use with the C++11 standard library's `<chrono>` facility, or with the Boost.Chrono library.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Examples

Performing a blocking wait (C++11):

```
// Construct a timer without setting an expiry time.  
asio::steady_timer timer(io_service);  
  
// Set an expiry time relative to now.  
timer.expires_from_now(std::chrono::seconds(5));  
  
// Wait for the timer to expire.  
timer.wait();
```

Performing an asynchronous wait (C++11):

```
void handler(const asio::error_code& error)  
{  
    if (!error)  
    {  
        // Timer expired.  
    }  
}  
  
...  
  
// Construct a timer with an absolute expiry time.  
asio::steady_timer timer(io_service,  
    std::chrono::steady_clock::now() + std::chrono::seconds(60));  
  
// Start an asynchronous wait.  
timer.async_wait(handler);
```

Changing an active waitable timer's expiry time

Changing the expiry time of a timer while there are pending asynchronous waits causes those wait operations to be cancelled. To ensure that the action associated with the timer is performed only once, use something like this: used:

```
void on_some_event()  
{  
    if (my_timer.expires_from_now(seconds(5)) > 0)  
    {  
        // We managed to cancel the timer. Start new asynchronous wait.  
        my_timer.async_wait(on_timeout);  
    }  
    else  
    {  
        // Too late, timer has already expired!  
    }  
}
```

```

void on_timeout(const asio::error_code& e)
{
    if (e != asio::error::operation_aborted)
    {
        // Timer was not cancelled, take necessary action.
    }
}

```

- The `asio::basic_waitable_timer::expires_from_now()` function cancels any pending asynchronous waits, and returns the number of asynchronous waits that were cancelled. If it returns 0 then you were too late and the wait handler has already been executed, or will soon be executed. If it returns 1 then the wait handler was successfully cancelled.
- If a wait handler is cancelled, the `error_code` passed to it contains the value `asio::error::operation_aborted`.

This typedef uses the C++11 `<chrono>` standard library facility, if available. Otherwise, it may use the Boost.Chrono library. To explicitly utilise Boost.Chrono, use the `basic_waitable_timer` template directly:

```
typedef basic_waitable_timer<boost::chrono::system_clock> timer;
```

Requirements

Header: `asio/system_timer.hpp`

Convenience header: None

5.212 thread

A simple abstraction for starting threads.

```
class thread :
    noncopyable
```

Member Functions

Name	Description
<code>join</code>	Wait for the thread to exit.
<code>thread</code>	Start a new thread that executes the supplied function.
<code>~thread</code>	Destructor.

The `thread` class implements the smallest possible subset of the functionality of `boost::thread`. It is intended to be used only for starting a thread and waiting for it to exit. If more extensive threading capabilities are required, you are strongly advised to use something else.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Example

A typical use of `thread` would be to launch a thread to run an `io_service`'s event processing loop:

```
asio::io_service io_service;
// ...
asio::thread t(boost::bind(&asio::io_service::run, &io_service));
// ...
t.join();
```

Requirements

Header: `asio/thread.hpp`

Convenience header: `asio.hpp`

5.212.1 `thread::join`

Wait for the thread to exit.

```
void join();
```

This function will block until the thread has exited.

If this function is not called before the thread object is destroyed, the thread itself will continue to run until completion. You will, however, no longer have the ability to wait for it to exit.

5.212.2 `thread::thread`

Start a new thread that executes the supplied function.

```
template<
    typename Function>
thread(
    Function f);
```

This constructor creates a new thread that will execute the given function or function object.

Parameters

f The function or function object to be run in the thread. The function signature must be:

```
void f();
```

5.212.3 `thread::~thread`

Destructor.

```
~thread();
```

5.213 `time_traits< boost::posix_time::ptime >`

Time traits specialised for `posix_time`.

```
template<>
struct time_traits< boost::posix_time::ptime >
```

Types

Name	Description
duration_type	The duration type.
time_type	The time type.

Member Functions

Name	Description
add	Add a duration to a time.
less_than	Test whether one time is less than another.
now	Get the current time.
subtract	Subtract one time from another.
to_posix_duration	Convert to POSIX duration type.

Requirements

Header: asio/time_traits.hpp

Convenience header: asio.hpp

5.213.1 time_traits< boost::posix_time::ptime >::add

Add a duration to a time.

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

5.213.2 time_traits< boost::posix_time::ptime >::duration_type

The duration type.

```
typedef boost::posix_time::time_duration duration_type;
```

Requirements

Header: asio/time_traits.hpp

Convenience header: asio.hpp

5.213.3 time_traits< boost::posix_time::ptime >::less_than

Test whether one time is less than another.

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

5.213.4 `time_traits< boost::posix_time::ptime >::now`

Get the current time.

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

5.213.5 `time_traits< boost::posix_time::ptime >::subtract`

Subtract one time from another.

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

5.213.6 `time_traits< boost::posix_time::ptime >::time_type`

The time type.

```
typedef boost::posix_time::ptime time_type;
```

Requirements

Header: asio/time_traits.hpp

Convenience header: asio.hpp

5.213.7 `time_traits< boost::posix_time::ptime >::to_posix_duration`

Convert to POSIX duration type.

```
static boost::posix_time::time_duration to_posix_duration(
    const duration_type & d);
```

5.214 transfer_all

Return a completion condition function object that indicates that a read or write operation should continue until all of the data has been transferred, or until an error occurs.

```
unspecified transfer_all();
```

This function is used to create an object, of unspecified type, that meets CompletionCondition requirements.

Example

Reading until a buffer is full:

```
boost::array<char, 128> buf;
asio::error_code ec;
std::size_t n = asio::read(
    sock, asio::buffer(buf),
    asio::transfer_all(), ec);
if (ec)
{
    // An error occurred.
```

```
    }
else
{
    // n == 128
}
```

Requirements

Header: asio/completion_condition.hpp

Convenience header: asio.hpp

5.215 transfer_at_least

Return a completion condition function object that indicates that a read or write operation should continue until a minimum number of bytes has been transferred, or until an error occurs.

```
unspecified transfer_at_least(
    std::size_t minimum);
```

This function is used to create an object, of unspecified type, that meets CompletionCondition requirements.

Example

Reading until a buffer is full or contains at least 64 bytes:

```
boost::array<char, 128> buf;
asio::error_code ec;
std::size_t n = asio::read(
    sock, asio::buffer(buf),
    asio::transfer_at_least(64), ec);
if (ec)
{
    // An error occurred.
}
else
{
    // n >= 64 && n <= 128
}
```

Requirements

Header: asio/completion_condition.hpp

Convenience header: asio.hpp

5.216 transfer_exactly

Return a completion condition function object that indicates that a read or write operation should continue until an exact number of bytes has been transferred, or until an error occurs.

```
unspecified transfer_exactly(
    std::size_t size);
```

This function is used to create an object, of unspecified type, that meets CompletionCondition requirements.

Example

Reading until a buffer is full or contains exactly 64 bytes:

```
boost::array<char, 128> buf;
asio::error_code ec;
std::size_t n = asio::read(
    sock, asio::buffer(buf),
    asio::transfer_exactly(64), ec);
if (ec)
{
    // An error occurred.
}
else
{
    // n == 64
}
```

Requirements

Header: asio/completion_condition.hpp

Convenience header: asio.hpp

5.217 use_future

A special value, similar to std::nothrow.

```
constexpr use_future_t use_future;
```

See the documentation for [use_future_t](#) for a usage example.

Requirements

Header: asio/use_future.hpp

Convenience header: None

5.218 use_future_t

Class used to specify that an asynchronous operation should return a future.

```
template<
    typename Allocator = std::allocator<void>>
class use_future_t
```

Types

Name	Description
allocator_type	The allocator type. The allocator is used when constructing the std::promise object for a given asynchronous operation.

Member Functions

Name	Description
get_allocator	Obtain allocator.
operator[]	Specify an alternate allocator.
use_future_t	Construct using default-constructed allocator. Construct using specified allocator.

The `use_future_t` class is used to indicate that an asynchronous operation should return a `std::future` object. A `use_future_t` object may be passed as a handler to an asynchronous operation, typically using the special value `asio::use_future`. For example:

```
std::future<std::size_t> my_future
= my_socket.async_read_some(my_buffer, asio::use_future);
```

The initiating function (`async_read_some` in the above example) returns a future that will receive the result of the operation. If the operation completes with an `error_code` indicating failure, it is converted into a `system_error` and passed back to the caller via the future.

Requirements

Header: `asio/use_future.hpp`

Convenience header: None

5.218.1 `use_future_t::allocator_type`

The allocator type. The allocator is used when constructing the `std::promise` object for a given asynchronous operation.

```
typedef Allocator allocator_type;
```

Requirements

Header: `asio/use_future.hpp`

Convenience header: None

5.218.2 `use_future_t::get_allocator`

Obtain allocator.

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

5.218.3 `use_future_t::operator[]`

Specify an alternate allocator.

```
template<
    typename OtherAllocator>
use_future_t< OtherAllocator > operator[](
    const OtherAllocator & allocator) const;
```

5.218.4 `use_future_t::use_future_t`

Construct using default-constructed allocator.

```
constexpr use_future_t();
```

Construct using specified allocator.

```
explicit use_future_t(
    const Allocator & allocator);
```

5.218.4.1 `use_future_t::use_future_t (1 of 2 overloads)`

Construct using default-constructed allocator.

```
constexpr use_future_t();
```

5.218.4.2 `use_future_t::use_future_t (2 of 2 overloads)`

Construct using specified allocator.

```
use_future_t(
    const Allocator & allocator);
```

5.219 `use_service`

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

This function is used to locate a service object that corresponds to the given service type. If there is no existing implementation of the service, then the `io_service` will create a new instance of the service.

Parameters

`ios` The `io_service` object that owns the service.

Return Value

The service interface implementing the specified service type. Ownership of the service interface is not transferred to the caller.

Requirements

Header: `asio/io_service.hpp`

Convenience header: `asio.hpp`

5.220 wait_traits

Wait traits suitable for use with the `basic_waitable_timer` class template.

```
template<
    typename Clock>
struct wait_traits
```

Member Functions

Name	Description
<code>to_wait_duration</code>	Convert a clock duration into a duration used for waiting.

Requirements

Header: `asio/wait_traits.hpp`

Convenience header: `asio.hpp`

5.220.1 wait_traits::to_wait_duration

Convert a clock duration into a duration used for waiting.

```
static Clock::duration to_wait_duration(
    const typename Clock::duration & d);
```

Return Value

`d`.

5.221 waitable_timer_service

Default service implementation for a timer.

```
template<
    typename Clock,
    typename WaitTraits = asio::wait_traits<Clock>>
class waitable_timer_service :
    public io_service::service
```

Types

Name	Description
<code>clock_type</code>	The clock type.
<code>duration</code>	The duration type of the clock.
<code>implementation_type</code>	The implementation type of the waitable timer.

Name	Description
time_point	The time point type of the clock.
traits_type	The wait traits type.

Member Functions

Name	Description
async_wait	
cancel	Cancel any asynchronous wait operations associated with the timer.
cancel_one	Cancels one asynchronous wait operation associated with the timer.
construct	Construct a new timer implementation.
destroy	Destroy a timer implementation.
expires_at	Get the expiry time for the timer as an absolute time. Set the expiry time for the timer as an absolute time.
expires_from_now	Get the expiry time for the timer relative to now. Set the expiry time for the timer relative to now.
get_io_service	Get the io_service object that owns the service.
wait	
waitable_timer_service	Construct a new timer service for the specified io_service.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/waitable_timer_service.hpp

Convenience header: asio.hpp

5.221.1 waitable_timer_service::async_wait

```
template<
    typename WaitHandler>
void-or-deduced async_wait(
    implementation_type & impl,
    WaitHandler handler);
```

5.221.2 `waitable_timer_service::cancel`

Cancel any asynchronous wait operations associated with the timer.

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

5.221.3 `waitable_timer_service::cancel_one`

Cancels one asynchronous wait operation associated with the timer.

```
std::size_t cancel_one(
    implementation_type & impl,
    asio::error_code & ec);
```

5.221.4 `waitable_timer_service::clock_type`

The clock type.

```
typedef Clock clock_type;
```

Requirements

Header: asio/waitable_timer_service.hpp

Convenience header: asio.hpp

5.221.5 `waitable_timer_service::construct`

Construct a new timer implementation.

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

5.221.6 `waitable_timer_service::destroy`

Destroy a timer implementation.

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

5.221.7 `waitable_timer_service::duration`

The duration type of the clock.

```
typedef clock_type::duration duration;
```

Requirements

Header: asio/waitable_timer_service.hpp

Convenience header: asio.hpp

5.221.8 `waitable_timer_service::expires_at`

Get the expiry time for the timer as an absolute time.

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

Set the expiry time for the timer as an absolute time.

```
std::size_t expires_at(
    implementation_type & impl,
    const time_point & expiry_time,
    asio::error_code & ec);
```

5.221.8.1 `waitable_timer_service::expires_at (1 of 2 overloads)`

Get the expiry time for the timer as an absolute time.

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

5.221.8.2 `waitable_timer_service::expires_at (2 of 2 overloads)`

Set the expiry time for the timer as an absolute time.

```
std::size_t expires_at(
    implementation_type & impl,
    const time_point & expiry_time,
    asio::error_code & ec);
```

5.221.9 `waitable_timer_service::expires_from_now`

Get the expiry time for the timer relative to now.

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

Set the expiry time for the timer relative to now.

```
std::size_t expires_from_now(
    implementation_type & impl,
    const duration & expiry_time,
    asio::error_code & ec);
```

5.221.9.1 `waitable_timer_service::expires_from_now (1 of 2 overloads)`

Get the expiry time for the timer relative to now.

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

5.221.9.2 `waitable_timer_service::expires_from_now` (2 of 2 overloads)

Set the expiry time for the timer relative to now.

```
std::size_t expires_from_now(
    implementation_type & impl,
    const duration & expiry_time,
    asio::error_code & ec);
```

5.221.10 `waitable_timer_service::get_io_service`

Inherited from io_service.

Get the `io_service` object that owns the service.

```
asio::io_service & get_io_service();
```

5.221.11 `waitable_timer_service::id`

The unique service identifier.

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

5.221.12 `waitable_timer_service::implementation_type`

The implementation type of the waitable timer.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asio/waitable_timer_service.hpp

Convenience header: asio.hpp

5.221.13 `waitable_timer_service::time_point`

The time point type of the clock.

```
typedef clock_type::time_point time_point;
```

Requirements

Header: asio/waitable_timer_service.hpp

Convenience header: asio.hpp

5.221.14 `waitable_timer_service::traits_type`

The wait traits type.

```
typedef WaitTraits traits_type;
```

Requirements

Header: asio/waitable_timer_service.hpp

Convenience header: asio.hpp

5.221.15 `waitable_timer_service::wait`

```
void wait(
    implementation_type & impl,
    asio::error_code & ec);
```

5.221.16 `waitable_timer_service::waitable_timer_service`

Construct a new timer service for the specified `io_service`.

```
waitable_timer_service(
    asio::io_service & io_service);
```

5.222 `windows::basic_handle`

Provides Windows handle functionality.

```
template<
    typename HandleService>
class basic_handle :
    public basic_io_object< HandleService >
```

Types

Name	Description
<code>implementation_type</code>	The underlying implementation type of I/O object.
<code>lowest_layer_type</code>	A <code>basic_handle</code> is always the lowest layer.
<code>native_handle_type</code>	The native representation of a handle.
<code>native_type</code>	(Deprecated: Use <code>native_handle_type</code> .) The native representation of a handle.
<code>service_type</code>	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
<code>assign</code>	Assign an existing native handle to the handle.

Name	Description
basic_handle	Construct a basic_handle without opening it. Construct a basic_handle on an existing native handle. Move-construct a basic_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_handle from another.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_handle	Protected destructor to prevent deletion through this type.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_handle` class template provides the ability to wrap a Windows handle.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/windows/basic_handle.hpp

Convenience header: asio.hpp

5.222.1 windows::basic_handle::assign

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);

asio::error_code assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.222.1.1 windows::basic_handle::assign (1 of 2 overloads)

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);
```

5.222.1.2 windows::basic_handle::assign (2 of 2 overloads)

Assign an existing native handle to the handle.

```
asio::error_code assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.222.2 windows::basic_handle::basic_handle

Construct a [windows::basic_handle](#) without opening it.

```
explicit basic_handle(
    asio::io_service & io_service);
```

Construct a [windows::basic_handle](#) on an existing native handle.

```
basic_handle(
    asio::io_service & io_service,
    const native_handle_type & handle);
```

Move-construct a [windows::basic_handle](#) from another.

```
basic_handle(
    basic_handle && other);
```

5.222.2.1 windows::basic_handle::basic_handle (1 of 3 overloads)

Construct a `windows::basic_handle` without opening it.

```
basic_handle(
    asio::io_service & io_service);
```

This constructor creates a handle without opening it.

Parameters

io_service The `io_service` object that the handle will use to dispatch handlers for any asynchronous operations performed on the handle.

5.222.2.2 windows::basic_handle::basic_handle (2 of 3 overloads)

Construct a `windows::basic_handle` on an existing native handle.

```
basic_handle(
    asio::io_service & io_service,
    const native_handle_type & handle);
```

This constructor creates a handle object to hold an existing native handle.

Parameters

io_service The `io_service` object that the handle will use to dispatch handlers for any asynchronous operations performed on the handle.

handle A native handle.

Exceptions

`asio::system_error` Thrown on failure.

5.222.2.3 windows::basic_handle::basic_handle (3 of 3 overloads)

Move-construct a `windows::basic_handle` from another.

```
basic_handle(
    basic_handle && other);
```

This constructor moves a handle from one object to another.

Parameters

other The other `windows::basic_handle` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_handle(io_service&)` constructor.

5.222.3 windows::basic_handle::cancel

Cancel all asynchronous operations associated with the handle.

```
void cancel();  
  
asio::error_code cancel(  
    asio::error_code & ec);
```

5.222.3.1 windows::basic_handle::cancel (1 of 2 overloads)

Cancel all asynchronous operations associated with the handle.

```
void cancel();
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.222.3.2 windows::basic_handle::cancel (2 of 2 overloads)

Cancel all asynchronous operations associated with the handle.

```
asio::error_code cancel(  
    asio::error_code & ec);
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

5.222.4 windows::basic_handle::close

Close the handle.

```
void close();  
  
asio::error_code close(  
    asio::error_code & ec);
```

5.222.4.1 windows::basic_handle::close (1 of 2 overloads)

Close the handle.

```
void close();
```

This function is used to close the handle. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.222.4.2 windows::basic_handle::close (2 of 2 overloads)

Close the handle.

```
asio::error_code close(
    asio::error_code & ec);
```

This function is used to close the handle. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

`ec` Set to indicate what error occurred, if any.

5.222.5 windows::basic_handle::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();

const implementation_type & get_implementation() const;
```

5.222.5.1 windows::basic_handle::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.222.5.2 windows::basic_handle::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.222.6 windows::basic_handle::get_io_service

Inherited from basic_io_object.

Get the `io_service` associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the `io_service` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_service` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.222.7 windows::basic_handle::get_service

Get the service associated with the I/O object.

```
service_type & get_service();  
  
const service_type & get_service() const;
```

5.222.7.1 windows::basic_handle::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.222.7.2 windows::basic_handle::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.222.8 windows::basic_handle::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.222.9 windows::basic_handle::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/windows/basic_handle.hpp

Convenience header: asio.hpp

5.222.10 windows::basic_handle::is_open

Determine whether the handle is open.

```
bool is_open() const;
```

5.222.11 windows::basic_handle::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.222.11.1 windows::basic_handle::lowest_layer (1 of 2 overloads)

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a `windows::basic_handle` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.222.11.2 windows::basic_handle::lowest_layer (2 of 2 overloads)

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a `windows::basic_handle` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.222.12 windows::basic_handle::lowest_layer_type

A `windows::basic_handle` is always the lowest layer.

```
typedef basic_handle< HandleService > lowest_layer_type;
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_handle is always the lowest layer.
native_handle_type	The native representation of a handle.

Name	Description
native_type	(Deprecated: Use native_handle_type.) The native representation of a handle.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
basic_handle	Construct a basic_handle without opening it. Construct a basic_handle on an existing native handle. Move-construct a basic_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_handle from another.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_handle	Protected destructor to prevent deletion through this type.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_handle` class template provides the ability to wrap a Windows handle.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/basic_handle.hpp`

Convenience header: `asio.hpp`

5.222.13 windows::basic_handle::native

(Deprecated: Use `native_handle()`.) Get the native handle representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the handle. This is intended to allow access to native handle functionality that is not otherwise provided.

5.222.14 windows::basic_handle::native_handle

Get the native handle representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the handle. This is intended to allow access to native handle functionality that is not otherwise provided.

5.222.15 windows::basic_handle::native_handle_type

The native representation of a handle.

```
typedef HandleService::native_handle_type native_handle_type;
```

Requirements

Header: `asio/windows/basic_handle.hpp`

Convenience header: `asio.hpp`

5.222.16 windows::basic_handle::native_type

(Deprecated: Use native_handle_type.) The native representation of a handle.

```
typedef HandleService::native_handle_type native_type;
```

Requirements

Header: asio/windows/basic_handle.hpp

Convenience header: asio.hpp

5.222.17 windows::basic_handle::operator=

Move-assign a windows::basic_handle from another.

```
basic_handle & operator=(  
    basic_handle && other);
```

This assignment operator moves a handle from one object to another.

Parameters

other The other windows::basic_handle object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the basic_handle(io_service&) constructor.

5.222.18 windows::basic_handle::service

Inherited from basic_io_object.

(Deprecated: Use get_service() .) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.222.19 windows::basic_handle::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef HandleService service_type;
```

Requirements

Header: asio/windows/basic_handle.hpp

Convenience header: asio.hpp

5.222.20 windows::basic_handle::~basic_handle

Protected destructor to prevent deletion through this type.

```
~basic_handle();
```

5.223 windows::basic_object_handle

Provides object-oriented handle functionality.

```
template<
    typename ObjectHandleService = object_handle_service>
class basic_object_handle :
    public windows::basic_handle< ObjectHandleService >
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_handle is always the lowest layer.
native_handle_type	The native representation of a handle.
native_type	(Deprecated: Use native_handle_type.) The native representation of a handle.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
async_wait	Start an asynchronous wait on the object handle.
basic_object_handle	Construct a basic_object_handle without opening it. Construct a basic_object_handle on an existing native handle. Move-construct a basic_object_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.

Name	Description
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_object_handle from another.
wait	Perform a blocking wait on the object handle.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_object_handle` class template provides asynchronous and blocking object-oriented handle functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/basic_object_handle.hpp`

Convenience header: `asio.hpp`

5.223.1 windows::basic_object_handle::assign

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);

asio::error_code assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.223.1.1 windows::basic_object_handle::assign (1 of 2 overloads)

Inherited from windows::basic_handle.

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);
```

5.223.1.2 windows::basic_object_handle::assign (2 of 2 overloads)

Inherited from windows::basic_handle.

Assign an existing native handle to the handle.

```
asio::error_code assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.223.2 windows::basic_object_handle::async_wait

Start an asynchronous wait on the object handle.

```
template<
    typename WaitHandler>
void-or-deduced async_wait(
    WaitHandler handler);
```

This function is be used to initiate an asynchronous wait against the object handle. It always returns immediately.

Parameters

handler The handler to be called when the object handle is set to the signalled state. Copies will be made of the handler as required.

The function signature of the handler must be:

```
void handler(
    const asio::error_code& error // Result of operation.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

5.223.3 windows::basic_object_handle::basic_object_handle

Construct a windows::basic_object_handle without opening it.

```
explicit basic_object_handle(
    asio::io_service & io_service);
```

Construct a windows::basic_object_handle on an existing native handle.

```
basic_object_handle(
    asio::io_service & io_service,
    const native_handle_type & native_handle);
```

Move-construct a windows::basic_object_handle from another.

```
basic_object_handle(
    basic_object_handle && other);
```

5.223.3.1 windows::basic_object_handle::basic_object_handle (1 of 3 overloads)

Construct a windows::basic_object_handle without opening it.

```
basic_object_handle(
    asio::io_service & io_service);
```

This constructor creates an object handle without opening it.

Parameters

io_service The `io_service` object that the object handle will use to dispatch handlers for any asynchronous operations performed on the handle.

5.223.3.2 windows::basic_object_handle::basic_object_handle (2 of 3 overloads)

Construct a windows::basic_object_handle on an existing native handle.

```
basic_object_handle(
    asio::io_service & io_service,
    const native_handle_type & native_handle);
```

This constructor creates an object handle object to hold an existing native handle.

Parameters

io_service The `io_service` object that the object handle will use to dispatch handlers for any asynchronous operations performed on the handle.

native_handle The new underlying handle implementation.

Exceptions

asio::system_error Thrown on failure.

5.223.3.3 windows::basic_object_handle::basic_object_handle (3 of 3 overloads)

Move-construct a windows::basic_object_handle from another.

```
basic_object_handle(
    basic_object_handle && other);
```

This constructor moves an object handle from one object to another.

Parameters

other The other windows::basic_object_handle object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the basic_object_handle(io_service&) constructor.

5.223.4 windows::basic_object_handle::cancel

Cancel all asynchronous operations associated with the handle.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.223.4.1 windows::basic_object_handle::cancel (1 of 2 overloads)

Inherited from windows::basic_handle.

Cancel all asynchronous operations associated with the handle.

```
void cancel();
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the asio::error::operation_aborted error.

Exceptions

asio::system_error Thrown on failure.

5.223.4.2 windows::basic_object_handle::cancel (2 of 2 overloads)

Inherited from windows::basic_handle.

Cancel all asynchronous operations associated with the handle.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the asio::error::operation_aborted error.

Parameters

ec Set to indicate what error occurred, if any.

5.223.5 windows::basic_object_handle::close

Close the handle.

```
void close();  
  
asio::error_code close(  
    asio::error_code & ec);
```

5.223.5.1 windows::basic_object_handle::close (1 of 2 overloads)

Inherited from windows::basic_handle.

Close the handle.

```
void close();
```

This function is used to close the handle. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.223.5.2 windows::basic_object_handle::close (2 of 2 overloads)

Inherited from windows::basic_handle.

Close the handle.

```
asio::error_code close(  
    asio::error_code & ec);
```

This function is used to close the handle. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

5.223.6 windows::basic_object_handle::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.223.6.1 windows::basic_object_handle::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.223.6.2 windows::basic_object_handle::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.223.7 windows::basic_object_handle::get_io_service

Inherited from basic_io_object.

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the **io_service** object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the **io_service** object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.223.8 windows::basic_object_handle::get_service

Get the service associated with the I/O object.

```
service_type & get_service();
```

```
const service_type & get_service() const;
```

5.223.8.1 windows::basic_object_handle::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.223.8.2 windows::basic_object_handle::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.223.9 windows::basic_object_handle::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.223.10 windows::basic_object_handle::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/windows/basic_object_handle.hpp

Convenience header: asio.hpp

5.223.11 windows::basic_object_handle::is_open

Inherited from windows::basic_handle.

Determine whether the handle is open.

```
bool is_open() const;
```

5.223.12 windows::basic_object_handle::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.223.12.1 windows::basic_object_handle::lowest_layer (1 of 2 overloads)

Inherited from windows::basic_handle.

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a `windows::basic_handle` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.223.12.2 windows::basic_object_handle::lowest_layer (2 of 2 overloads)

Inherited from windows::basic_handle.

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a windows::basic_handle cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.223.13 windows::basic_object_handle::lowest_layer_type

Inherited from windows::basic_handle.

A windows::basic_handle is always the lowest layer.

```
typedef basic_handle< ObjectHandleService > lowest_layer_type;
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_handle is always the lowest layer.
native_handle_type	The native representation of a handle.
native_type	(Deprecated: Use native_handle_type.) The native representation of a handle.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
basic_handle	Construct a basic_handle without opening it. Construct a basic_handle on an existing native handle. Move-construct a basic_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.

Name	Description
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_handle from another.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_handle	Protected destructor to prevent deletion through this type.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_handle` class template provides the ability to wrap a Windows handle.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/basic_object_handle.hpp`

Convenience header: `asio.hpp`

5.223.14 windows::basic_object_handle::native

Inherited from windows::basic_handle.

(Deprecated: Use native_handle().) Get the native handle representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the handle. This is intended to allow access to native handle functionality that is not otherwise provided.

5.223.15 windows::basic_object_handle::native_handle

Inherited from windows::basic_handle.

Get the native handle representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the handle. This is intended to allow access to native handle functionality that is not otherwise provided.

5.223.16 windows::basic_object_handle::native_handle_type

The native representation of a handle.

```
typedef ObjectHandleService::native_handle_type native_handle_type;
```

Requirements

Header: asio/windows/basic_object_handle.hpp

Convenience header: asio.hpp

5.223.17 windows::basic_object_handle::native_type

Inherited from windows::basic_handle.

(Deprecated: Use native_handle_type.) The native representation of a handle.

```
typedef ObjectHandleService::native_handle_type native_type;
```

Requirements

Header: asio/windows/basic_object_handle.hpp

Convenience header: asio.hpp

5.223.18 windows::basic_object_handle::operator=

Move-assign a windows::basic_object_handle from another.

```
basic_object_handle & operator=(  
    basic_object_handle && other);
```

This assignment operator moves an object handle from one object to another.

Parameters

other The other `windows::basic_object_handle` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_object_handle(io_service&)` constructor.

5.223.19 windows::basic_object_handle::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`.) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.223.20 windows::basic_object_handle::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef ObjectHandleService service_type;
```

Requirements

Header: `asio/windows/basic_object_handle.hpp`

Convenience header: `asio.hpp`

5.223.21 windows::basic_object_handle::wait

Perform a blocking wait on the object handle.

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

5.223.21.1 windows::basic_object_handle::wait (1 of 2 overloads)

Perform a blocking wait on the object handle.

```
void wait();
```

This function is used to wait for the object handle to be set to the signalled state. This function blocks and does not return until the object handle has been set to the signalled state.

Exceptions

`asio::system_error` Thrown on failure.

5.223.21.2 windows::basic_object_handle::wait (2 of 2 overloads)

Perform a blocking wait on the object handle.

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

This function is used to wait for the object handle to be set to the signalled state. This function blocks and does not return until the object handle has been set to the signalled state.

Parameters

`ec` Set to indicate what error occurred, if any.

5.224 windows::basic_random_access_handle

Provides random-access handle functionality.

```
template<
    typename RandomAccessHandleService = random_access_handle_service>
class basic_random_access_handle :
    public windows::basic_handle< RandomAccessHandleService >
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_handle is always the lowest layer.
native_handle_type	The native representation of a handle.
native_type	(Deprecated: Use native_handle_type.) The native representation of a handle.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
async_read_some_at	Start an asynchronous read at the specified offset.

Name	Description
async_write_some_at	Start an asynchronous write at the specified offset.
basic_random_access_handle	Construct a basic_random_access_handle without opening it. Construct a basic_random_access_handle on an existing native handle. Move-construct a basic_random_access_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_random_access_handle from another.
read_some_at	Read some data from the handle at the specified offset.
write_some_at	Write some data to the handle at the specified offset.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_random_access_handle` class template provides asynchronous and blocking random-access handle functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/windows/basic_random_access_handle.hpp

Convenience header: asio.hpp

5.224.1 windows::basic_random_access_handle::assign

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);

asio::error_code assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.224.1.1 windows::basic_random_access_handle::assign (1 of 2 overloads)

Inherited from windows::basic_handle.

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);
```

5.224.1.2 windows::basic_random_access_handle::assign (2 of 2 overloads)

Inherited from windows::basic_handle.

Assign an existing native handle to the handle.

```
asio::error_code assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.224.2 windows::basic_random_access_handle::async_read_some_at

Start an asynchronous read at the specified offset.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some_at (
    uint64_t offset,
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

This function is used to asynchronously read data from the random-access handle. The function call always returns immediately.

Parameters

offset The offset at which the data will be read.

buffers One or more buffers into which the data will be read. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes read.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The read operation may not read all of the requested number of bytes. Consider using the `async_read_at` function if you need to ensure that the requested amount of data is read before the asynchronous operation completes.

Example

To read into a single data buffer use the `buffer` function as follows:

```
handle.async_read_some_at(42, asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.224.3 windows::basic_random_access_handle::async_write_some_at

Start an asynchronous write at the specified offset.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some_at(
    uint64_t offset,
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

This function is used to asynchronously write data to the random-access handle. The function call always returns immediately.

Parameters

offset The offset at which the data will be written.

buffers One or more data buffers to be written to the handle. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes written.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The write operation may not transmit all of the data to the peer. Consider using the `async_write_at` function if you need to ensure that all data is written before the asynchronous operation completes.

Example

To write a single data buffer use the `buffer` function as follows:

```
handle.async_write_some_at(42, asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.224.4 windows::basic_random_access_handle::basic_random_access_handle

Construct a `windows::basic_random_access_handle` without opening it.

```
explicit basic_random_access_handle(
    asio::io_service & io_service);
```

Construct a `windows::basic_random_access_handle` on an existing native handle.

```
basic_random_access_handle(
    asio::io_service & io_service,
    const native_handle_type & handle);
```

Move-construct a `windows::basic_random_access_handle` from another.

```
basic_random_access_handle(
    basic_random_access_handle && other);
```

5.224.4.1 windows::basic_random_access_handle::basic_random_access_handle (1 of 3 overloads)

Construct a `windows::basic_random_access_handle` without opening it.

```
basic_random_access_handle(
    asio::io_service & io_service);
```

This constructor creates a random-access handle without opening it. The handle needs to be opened before data can be written to or read from it.

Parameters

io_service The `io_service` object that the random-access handle will use to dispatch handlers for any asynchronous operations performed on the handle.

5.224.4.2 windows::basic_random_access_handle::basic_random_access_handle (2 of 3 overloads)

Construct a windows::basic_random_access_handle on an existing native handle.

```
basic_random_access_handle(
    asio::io_service & io_service,
    const native_handle_type & handle);
```

This constructor creates a random-access handle object to hold an existing native handle.

Parameters

io_service The **io_service** object that the random-access handle will use to dispatch handlers for any asynchronous operations performed on the handle.

handle The new underlying handle implementation.

Exceptions

asio::system_error Thrown on failure.

5.224.4.3 windows::basic_random_access_handle::basic_random_access_handle (3 of 3 overloads)

Move-construct a windows::basic_random_access_handle from another.

```
basic_random_access_handle(
    basic_random_access_handle && other);
```

This constructor moves a random-access handle from one object to another.

Parameters

other The other windows::basic_random_access_handle object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the basic_random_access_handle(io_service&) constructor.

5.224.5 windows::basic_random_access_handle::cancel

Cancel all asynchronous operations associated with the handle.

```
void cancel();

asio::error_code cancel(
    asio::error_code & ec);
```

5.224.5.1 windows::basic_random_access_handle::cancel (1 of 2 overloads)

Inherited from windows::basic_handle.

Cancel all asynchronous operations associated with the handle.

```
void cancel();
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.224.5.2 windows::basic_random_access_handle::cancel (2 of 2 overloads)

Inherited from windows::basic_handle.

Cancel all asynchronous operations associated with the handle.

```
asio::error_code cancel(
    asio::error_code & ec);
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

`ec` Set to indicate what error occurred, if any.

5.224.6 windows::basic_random_access_handle::close

Close the handle.

```
void close();
```

```
asio::error_code close(
    asio::error_code & ec);
```

5.224.6.1 windows::basic_random_access_handle::close (1 of 2 overloads)

Inherited from windows::basic_handle.

Close the handle.

```
void close();
```

This function is used to close the handle. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.224.6.2 windows::basic_random_access_handle::close (2 of 2 overloads)

Inherited from windows::basic_handle.

Close the handle.

```
asio::error_code close(  
    asio::error_code & ec);
```

This function is used to close the handle. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

ec Set to indicate what error occurred, if any.

5.224.7 windows::basic_random_access_handle::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.224.7.1 windows::basic_random_access_handle::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.224.7.2 windows::basic_random_access_handle::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.224.8 windows::basic_random_access_handle::get_io_service

Inherited from basic_io_object.

Get the `io_service` associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the `io_service` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_service` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.224.9 windows::basic_random_access_handle::get_service

Get the service associated with the I/O object.

```
service_type & get_service();  
  
const service_type & get_service() const;
```

5.224.9.1 windows::basic_random_access_handle::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.224.9.2 windows::basic_random_access_handle::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.224.10 windows::basic_random_access_handle::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.224.11 windows::basic_random_access_handle::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/windows/basic_random_access_handle.hpp

Convenience header: asio.hpp

5.224.12 windows::basic_random_access_handle::is_open

Inherited from windows::basic_handle.

Determine whether the handle is open.

```
bool is_open() const;
```

5.224.13 windows::basic_random_access_handle::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.224.13.1 windows::basic_random_access_handle::lowest_layer (1 of 2 overloads)

Inherited from windows::basic_handle.

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a [windows::basic_handle](#) cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.224.13.2 windows::basic_random_access_handle::lowest_layer (2 of 2 overloads)

Inherited from windows::basic_handle.

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a [windows::basic_handle](#) cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.224.14 windows::basic_random_access_handle::lowest_layer_type

Inherited from windows::basic_handle.

A [windows::basic_handle](#) is always the lowest layer.

```
typedef basic_handle< RandomAccessHandleService > lowest_layer_type;
```

Name	Description
------	-------------

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_handle is always the lowest layer.
native_handle_type	The native representation of a handle.
native_type	(Deprecated: Use native_handle_type.) The native representation of a handle.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
basic_handle	Construct a basic_handle without opening it. Construct a basic_handle on an existing native handle. Move-construct a basic_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_handle from another.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_handle	Protected destructor to prevent deletion through this type.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_handle` class template provides the ability to wrap a Windows handle.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/basic_random_access_handle.hpp`

Convenience header: `asio.hpp`

5.224.15 windows::basic_random_access_handle::native

Inherited from windows::basic_handle.

(Deprecated: Use `native_handle()`.) Get the native handle representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the handle. This is intended to allow access to native handle functionality that is not otherwise provided.

5.224.16 windows::basic_random_access_handle::native_handle

Inherited from windows::basic_handle.

Get the native handle representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the handle. This is intended to allow access to native handle functionality that is not otherwise provided.

5.224.17 windows::basic_random_access_handle::native_handle_type

The native representation of a handle.

```
typedef RandomAccessHandleService::native_handle_type native_handle_type;
```

Requirements

Header: asio/windows/basic_random_access_handle.hpp

Convenience header: asio.hpp

5.224.18 windows::basic_random_access_handle::native_type

(Deprecated: Use native_handle_type.) The native representation of a handle.

```
typedef RandomAccessHandleService::native_handle_type native_type;
```

Requirements

Header: asio/windows/basic_random_access_handle.hpp

Convenience header: asio.hpp

5.224.19 windows::basic_random_access_handle::operator=

Move-assign a `windows::basic_random_access_handle` from another.

```
basic_random_access_handle & operator=(  
    basic_random_access_handle && other);
```

This assignment operator moves a random-access handle from one object to another.

Parameters

other The other `windows::basic_random_access_handle` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_random_access_handle(io_service&)` constructor.

5.224.20 windows::basic_random_access_handle::read_some_at

Read some data from the handle at the specified offset.

```
template<  
    typename MutableBufferSequence>  
std::size_t read_some_at(  
    uint64_t offset,  
    const MutableBufferSequence & buffers);  
  
template<
```

```
typename MutableBufferSequence>
std::size_t read_some_at(
    uint64_t offset,
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.224.20.1 windows::basic_random_access_handle::read_some_at (1 of 2 overloads)

Read some data from the handle at the specified offset.

```
template<
    typename MutableBufferSequence>
std::size_t read_some_at(
    uint64_t offset,
    const MutableBufferSequence & buffers);
```

This function is used to read data from the random-access handle. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

offset The offset at which the data will be read.

buffers One or more buffers into which the data will be read.

Return Value

The number of bytes read.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The `read_some` operation may not read all of the requested number of bytes. Consider using the `read_at` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

Example

To read into a single data buffer use the `buffer` function as follows:

```
handle.read_some_at(42, asio::buffer(data, size));
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.224.20.2 windows::basic_random_access_handle::read_some_at (2 of 2 overloads)

Read some data from the handle at the specified offset.

```
template<
    typename MutableBufferSequence>
std::size_t read_some_at(
    uint64_t offset,
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to read data from the random-access handle. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

offset The offset at which the data will be read.

buffers One or more buffers into which the data will be read.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes read. Returns 0 if an error occurred.

Remarks

The read_some operation may not read all of the requested number of bytes. Consider using the [read_at](#) function if you need to ensure that the requested amount of data is read before the blocking operation completes.

5.224.21 windows::basic_random_access_handle::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`.) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.224.22 windows::basic_random_access_handle::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef RandomAccessHandleService service_type;
```

Requirements

Header: `asio/windows/basic_random_access_handle.hpp`

Convenience header: `asio.hpp`

5.224.23 windows::basic_random_access_handle::write_some_at

Write some data to the handle at the specified offset.

```
template<
    typename ConstBufferSequence>
std::size_t write_some_at(
    uint64_t offset,
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
std::size_t write_some_at(
    uint64_t offset,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.224.23.1 windows::basic_random_access_handle::write_some_at (1 of 2 overloads)

Write some data to the handle at the specified offset.

```
template<
    typename ConstBufferSequence>
std::size_t write_some_at(
    uint64_t offset,
    const ConstBufferSequence & buffers);
```

This function is used to write data to the random-access handle. The function call will block until one or more bytes of the data has been written successfully, or until an error occurs.

Parameters

offset The offset at which the data will be written.

buffers One or more data buffers to be written to the handle.

Return Value

The number of bytes written.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The `write_some_at` operation may not write all of the data. Consider using the [write_at](#) function if you need to ensure that all data is written before the blocking operation completes.

Example

To write a single data buffer use the `buffer` function as follows:

```
handle.write_some_at(42, asio::buffer(data, size));
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.224.23.2 windows::basic_random_access_handle::write_some_at (2 of 2 overloads)

Write some data to the handle at the specified offset.

```
template<
    typename ConstBufferSequence>
std::size_t write_some_at(
    uint64_t offset,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to write data to the random-access handle. The function call will block until one or more bytes of the data has been written successfully, or until an error occurs.

Parameters

offset The offset at which the data will be written.

buffers One or more data buffers to be written to the handle.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes written. Returns 0 if an error occurred.

Remarks

The `write_some` operation may not transmit all of the data to the peer. Consider using the `write_at` function if you need to ensure that all data is written before the blocking operation completes.

5.225 windows::basic_stream_handle

Provides stream-oriented handle functionality.

```
template<
    typename StreamHandleService = stream_handle_service>
class basic_stream_handle :
    public windows::basic_handle< StreamHandleService >
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_handle is always the lowest layer.
native_handle_type	The native representation of a handle.
native_type	(Deprecated: Use native_handle_type.) The native representation of a handle.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
async_read_some	Start an asynchronous read.
async_write_some	Start an asynchronous write.
basic_stream_handle	Construct a basic_stream_handle without opening it. Construct a basic_stream_handle on an existing native handle. Move-construct a basic_stream_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_stream_handle from another.
read_some	Read some data from the handle.
write_some	Write some data to the handle.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_stream_handle` class template provides asynchronous and blocking stream-oriented handle functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/basic_stream_handle.hpp`

Convenience header: `asio.hpp`

5.225.1 windows::basic_stream_handle::assign

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);

asio::error_code assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.225.1.1 windows::basic_stream_handle::assign (1 of 2 overloads)

Inherited from windows::basic_handle.

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);
```

5.225.1.2 windows::basic_stream_handle::assign (2 of 2 overloads)

Inherited from windows::basic_handle.

Assign an existing native handle to the handle.

```
asio::error_code assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.225.2 windows::basic_stream_handle::async_read_some

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

This function is used to asynchronously read data from the stream handle. The function call always returns immediately.

Parameters

buffers One or more buffers into which the data will be read. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the read operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes read.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The read operation may not read all of the requested number of bytes. Consider using the `async_read` function if you need to ensure that the requested amount of data is read before the asynchronous operation completes.

Example

To read into a single data buffer use the `buffer` function as follows:

```
handle.async_read_some(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.225.3 windows::basic_stream_handle::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some(
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

This function is used to asynchronously write data to the stream handle. The function call always returns immediately.

Parameters

buffers One or more data buffers to be written to the handle. Although the buffers object may be copied as necessary, ownership of the underlying memory blocks is retained by the caller, which must guarantee that they remain valid until the handler is called.

handler The handler to be called when the write operation completes. Copies will be made of the handler as required. The function signature of the handler must be:

```
void handler(
    const asio::error_code& error, // Result of operation.
    std::size_t bytes_transferred           // Number of bytes written.
);
```

Regardless of whether the asynchronous operation completes immediately or not, the handler will not be invoked from within this function. Invocation of the handler will be performed in a manner equivalent to using `asio::io_service::post()`.

Remarks

The write operation may not transmit all of the data to the peer. Consider using the `async_write` function if you need to ensure that all data is written before the asynchronous operation completes.

Example

To write a single data buffer use the `buffer` function as follows:

```
handle.async_write_some(asio::buffer(data, size), handler);
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.225.4 windows::basic_stream_handle::basic_stream_handle

Construct a `windows::basic_stream_handle` without opening it.

```
explicit basic_stream_handle(
    asio::io_service & io_service);
```

Construct a `windows::basic_stream_handle` on an existing native handle.

```
basic_stream_handle(
    asio::io_service & io_service,
    const native_handle_type & handle);
```

Move-construct a `windows::basic_stream_handle` from another.

```
basic_stream_handle(
    basic_stream_handle && other);
```

5.225.4.1 windows::basic_stream_handle::basic_stream_handle (1 of 3 overloads)

Construct a `windows::basic_stream_handle` without opening it.

```
basic_stream_handle(
    asio::io_service & io_service);
```

This constructor creates a stream handle without opening it. The handle needs to be opened and then connected or accepted before data can be sent or received on it.

Parameters

io_service The `io_service` object that the stream handle will use to dispatch handlers for any asynchronous operations performed on the handle.

5.225.4.2 windows::basic_stream_handle::basic_stream_handle (2 of 3 overloads)

Construct a `windows::basic_stream_handle` on an existing native handle.

```
basic_stream_handle(
    asio::io_service & io_service,
    const native_handle_type & handle);
```

This constructor creates a stream handle object to hold an existing native handle.

Parameters

io_service The `io_service` object that the stream handle will use to dispatch handlers for any asynchronous operations performed on the handle.

handle The new underlying handle implementation.

Exceptions

`asio::system_error` Thrown on failure.

5.225.4.3 windows::basic_stream_handle::basic_stream_handle (3 of 3 overloads)

Move-construct a `windows::basic_stream_handle` from another.

```
basic_stream_handle(
    basic_stream_handle && other);
```

This constructor moves a stream handle from one object to another.

Parameters

other The other `windows::basic_stream_handle` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_stream_handle(io_serv ice&)` constructor.

5.225.5 windows::basic_stream_handle::cancel

Cancel all asynchronous operations associated with the handle.

```
void cancel();  
  
asio::error_code cancel(  
    asio::error_code & ec);
```

5.225.5.1 windows::basic_stream_handle::cancel (1 of 2 overloads)

Inherited from windows::basic_handle.

Cancel all asynchronous operations associated with the handle.

```
void cancel();
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.225.5.2 windows::basic_stream_handle::cancel (2 of 2 overloads)

Inherited from windows::basic_handle.

Cancel all asynchronous operations associated with the handle.

```
asio::error_code cancel(  
    asio::error_code & ec);
```

This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error.

Parameters

`ec` Set to indicate what error occurred, if any.

5.225.6 windows::basic_stream_handle::close

Close the handle.

```
void close();  
  
asio::error_code close(  
    asio::error_code & ec);
```

5.225.6.1 windows::basic_stream_handle::close (1 of 2 overloads)

Inherited from windows::basic_handle.

Close the handle.

```
void close();
```

This function is used to close the handle. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Exceptions

`asio::system_error` Thrown on failure.

5.225.6.2 windows::basic_stream_handle::close (2 of 2 overloads)

Inherited from windows::basic_handle.

Close the handle.

```
asio::error_code close(
    asio::error_code & ec);
```

This function is used to close the handle. Any asynchronous read or write operations will be cancelled immediately, and will complete with the `asio::error::operation_aborted` error.

Parameters

`ec` Set to indicate what error occurred, if any.

5.225.7 windows::basic_stream_handle::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();

const implementation_type & get_implementation() const;
```

5.225.7.1 windows::basic_stream_handle::get_implementation (1 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.225.7.2 windows::basic_stream_handle::get_implementation (2 of 2 overloads)

Inherited from basic_io_object.

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.225.8 windows::basic_stream_handle::get_io_service

Inherited from basic_io_object.

Get the **io_service** associated with the object.

```
asio::io_service & get_io_service();
```

This function may be used to obtain the **io_service** object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the **io_service** object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.225.9 windows::basic_stream_handle::get_service

Get the service associated with the I/O object.

```
service_type & get_service();  
  
const service_type & get_service() const;
```

5.225.9.1 windows::basic_stream_handle::get_service (1 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.225.9.2 windows::basic_stream_handle::get_service (2 of 2 overloads)

Inherited from basic_io_object.

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.225.10 windows::basic_stream_handle::implementation

Inherited from basic_io_object.

(Deprecated: Use `get_implementation()`.) The underlying implementation of the I/O object.

```
implementation_type implementation;
```

5.225.11 windows::basic_stream_handle::implementation_type

Inherited from basic_io_object.

The underlying implementation type of I/O object.

```
typedef service_type::implementation_type implementation_type;
```

Requirements

Header: asio/windows/basic_stream_handle.hpp

Convenience header: asio.hpp

5.225.12 windows::basic_stream_handle::is_open

Inherited from windows::basic_handle.

Determine whether the handle is open.

```
bool is_open() const;
```

5.225.13 windows::basic_stream_handle::lowest_layer

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.225.13.1 windows::basic_stream_handle::lowest_layer (1 of 2 overloads)

Inherited from windows::basic_handle.

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

This function returns a reference to the lowest layer in a stack of layers. Since a `windows::basic_handle` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.225.13.2 windows::basic_stream_handle::lowest_layer (2 of 2 overloads)

Inherited from windows::basic_handle.

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

This function returns a const reference to the lowest layer in a stack of layers. Since a `windows::basic_handle` cannot contain any further layers, it simply returns a reference to itself.

Return Value

A const reference to the lowest layer in the stack of layers. Ownership is not transferred to the caller.

5.225.14 windows::basic_stream_handle::lowest_layer_type

Inherited from windows::basic_handle.

A windows::basic_handle is always the lowest layer.

```
typedef basic_handle< StreamHandleService > lowest_layer_type;
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_handle is always the lowest layer.
native_handle_type	The native representation of a handle.
native_type	(Deprecated: Use native_handle_type.) The native representation of a handle.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
basic_handle	Construct a basic_handle without opening it. Construct a basic_handle on an existing native handle. Move-construct a basic_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_handle from another.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.
~basic_handle	Protected destructor to prevent deletion through this type.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_handle` class template provides the ability to wrap a Windows handle.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/basic_stream_handle.hpp`

Convenience header: `asio.hpp`

5.225.15 windows::basic_stream_handle::native

Inherited from windows::basic_handle.

(Deprecated: Use `native_handle()`) Get the native handle representation.

```
native_type native();
```

This function may be used to obtain the underlying representation of the handle. This is intended to allow access to native handle functionality that is not otherwise provided.

5.225.16 windows::basic_stream_handle::native_handle

Inherited from windows::basic_handle.

Get the native handle representation.

```
native_handle_type native_handle();
```

This function may be used to obtain the underlying representation of the handle. This is intended to allow access to native handle functionality that is not otherwise provided.

5.225.17 windows::basic_stream_handle::native_handle_type

The native representation of a handle.

```
typedef StreamHandleService::native_handle_type native_handle_type;
```

Requirements

Header: asio/windows/basic_stream_handle.hpp

Convenience header: asio.hpp

5.225.18 windows::basic_stream_handle::native_type

(Deprecated: Use native_handle_type.) The native representation of a handle.

```
typedef StreamHandleService::native_handle_type native_type;
```

Requirements

Header: asio/windows/basic_stream_handle.hpp

Convenience header: asio.hpp

5.225.19 windows::basic_stream_handle::operator=

Move-assign a `windows::basic_stream_handle` from another.

```
basic_stream_handle & operator=(  
    basic_stream_handle && other);
```

This assignment operator moves a stream handle from one object to another.

Parameters

other The other `windows::basic_stream_handle` object from which the move will occur.

Remarks

Following the move, the moved-from object is in the same state as if constructed using the `basic_stream_handle(io_service&)` constructor.

5.225.20 windows::basic_stream_handle::read_some

Read some data from the handle.

```
template<  
    typename MutableBufferSequence>  
std::size_t read_some(  
    const MutableBufferSequence & buffers);  
  
template<  
    typename MutableBufferSequence>  
std::size_t read_some(  
    const MutableBufferSequence & buffers,  
    asio::error_code & ec);
```

5.225.20.1 windows::basic_stream_handle::read_some (1 of 2 overloads)

Read some data from the handle.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers);
```

This function is used to read data from the stream handle. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be read.

Return Value

The number of bytes read.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The `read_some` operation may not read all of the requested number of bytes. Consider using the `read` function if you need to ensure that the requested amount of data is read before the blocking operation completes.

Example

To read into a single data buffer use the `buffer` function as follows:

```
handle.read_some(asio::buffer(data, size));
```

See the `buffer` documentation for information on reading into multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.225.20.2 windows::basic_stream_handle::read_some (2 of 2 overloads)

Read some data from the handle.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to read data from the stream handle. The function call will block until one or more bytes of data has been read successfully, or until an error occurs.

Parameters

buffers One or more buffers into which the data will be read.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes read. Returns 0 if an error occurred.

Remarks

The read_some operation may not read all of the requested number of bytes. Consider using the [read](#) function if you need to ensure that the requested amount of data is read before the blocking operation completes.

5.225.21 windows::basic_stream_handle::service

Inherited from basic_io_object.

(Deprecated: Use `get_service()`.) The service associated with the I/O object.

```
service_type & service;
```

Remarks

Available only for services that do not support movability.

5.225.22 windows::basic_stream_handle::service_type

Inherited from basic_io_object.

The type of the service that will be used to provide I/O operations.

```
typedef StreamHandleService service_type;
```

Requirements

Header: asio/windows/basic_stream_handle.hpp

Convenience header: asio.hpp

5.225.23 windows::basic_stream_handle::write_some

Write some data to the handle.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.225.23.1 windows::basic_stream_handle::write_some (1 of 2 overloads)

Write some data to the handle.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers);
```

This function is used to write data to the stream handle. The function call will block until one or more bytes of the data has been written successfully, or until an error occurs.

Parameters

buffers One or more data buffers to be written to the handle.

Return Value

The number of bytes written.

Exceptions

asio::system_error Thrown on failure. An error code of `asio::error::eof` indicates that the connection was closed by the peer.

Remarks

The `write_some` operation may not transmit all of the data to the peer. Consider using the [write](#) function if you need to ensure that all data is written before the blocking operation completes.

Example

To write a single data buffer use the [buffer](#) function as follows:

```
handle.write_some(asio::buffer(data, size));
```

See the [buffer](#) documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.225.23.2 windows::basic_stream_handle::write_some (2 of 2 overloads)

Write some data to the handle.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to write data to the stream handle. The function call will block until one or more bytes of the data has been written successfully, or until an error occurs.

Parameters

buffers One or more data buffers to be written to the handle.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes written. Returns 0 if an error occurred.

Remarks

The write_some operation may not transmit all of the data to the peer. Consider using the [write](#) function if you need to ensure that all data is written before the blocking operation completes.

5.226 windows::object_handle

Typedef for the typical usage of an object handle.

```
typedef basic_object_handle object_handle;
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_handle is always the lowest layer.
native_handle_type	The native representation of a handle.
native_type	(Deprecated: Use native_handle_type.) The native representation of a handle.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
async_wait	Start an asynchronous wait on the object handle.
basic_object_handle	Construct a basic_object_handle without opening it. Construct a basic_object_handle on an existing native handle. Move-construct a basic_object_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.

Name	Description
close	Close the handle.
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_object_handle from another.
wait	Perform a blocking wait on the object handle.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_object_handle` class template provides asynchronous and blocking object-oriented handle functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/object_handle.hpp`

Convenience header: `asio.hpp`

5.227 windows::object_handle_service

Default service implementation for an object handle.

```
class object_handle_service :  
    public io_service::service
```

Types

Name	Description
implementation_type	The type of an object handle implementation.
native_handle_type	The native handle type.

Member Functions

Name	Description
assign	Assign an existing native handle to an object handle.
async_wait	Start an asynchronous wait.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close an object handle implementation.
construct	Construct a new object handle implementation.
destroy	Destroy an object handle implementation.
get_io_service	Get the io_service object that owns the service.
is_open	Determine whether the handle is open.
move_assign	Move-assign from another object handle implementation.
move_construct	Move-construct a new object handle implementation.
native_handle	Get the native handle implementation.
object_handle_service	Construct a new object handle service for the specified io_service.
wait	

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/windows/object_handle_service.hpp

Convenience header: asio.hpp

5.227.1 windows::object_handle_service::assign

Assign an existing native handle to an object handle.

```
asio::error_code assign(
    implementation_type & impl,
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.227.2 windows::object_handle_service::async_wait

Start an asynchronous wait.

```
template<
    typename WaitHandler>
void-or-deduced async_wait(
    implementation_type & impl,
    WaitHandler handler);
```

5.227.3 windows::object_handle_service::cancel

Cancel all asynchronous operations associated with the handle.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.227.4 windows::object_handle_service::close

Close an object handle implementation.

```
asio::error_code close(
    implementation_type & impl,
    asio::error_code & ec);
```

5.227.5 windows::object_handle_service::construct

Construct a new object handle implementation.

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

5.227.6 windows::object_handle_service::destroy

Destroy an object handle implementation.

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

5.227.7 windows::object_handle_service::get_io_service

Inherited from io_service.

Get the `io_service` object that owns the service.

```
asio::io_service & get_io_service();
```

5.227.8 windows::object_handle_service::id

The unique service identifier.

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

5.227.9 windows::object_handle_service::implementation_type

The type of an object handle implementation.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: `asio/windows/object_handle_service.hpp`

Convenience header: `asio.hpp`

5.227.10 windows::object_handle_service::is_open

Determine whether the handle is open.

```
bool is_open(
    const implementation_type & impl) const;
```

5.227.11 windows::object_handle_service::move_assign

Move-assign from another object handle implementation.

```
void move_assign(
    implementation_type & impl,
    object_handle_service & other_service,
    implementation_type & other_impl);
```

5.227.12 windows::object_handle_service::move_construct

Move-construct a new object handle implementation.

```
void move_construct(
    implementation_type & impl,
    implementation_type & other_impl);
```

5.227.13 windows::object_handle_service::native_handle

Get the native handle implementation.

```
native_handle_type native_handle(
    implementation_type & impl);
```

5.227.14 windows::object_handle_service::native_handle_type

The native handle type.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/windows/object_handle_service.hpp

Convenience header: asio.hpp

5.227.15 windows::object_handle_service::object_handle_service

Construct a new object handle service for the specified [io_service](#).

```
object_handle_service(
    asio::io_service & io_service);
```

5.227.16 windows::object_handle_service::wait

```
void wait(
    implementation_type & impl,
    asio::error_code & ec);
```

5.228 windows::overlapped_ptr

Wraps a handler to create an OVERLAPPED object for use with overlapped I/O.

```
class overlapped_ptr :
    noncopyable
```

Member Functions

Name	Description
complete	Post completion notification for overlapped operation. Releases ownership.
get	Get the contained OVERLAPPED object.
overlapped_ptr	Construct an empty overlapped_ptr. Construct an overlapped_ptr to contain the specified handler.
release	Release ownership of the OVERLAPPED object.
reset	Reset to empty. Reset to contain the specified handler, freeing any current OVERLAPPED object.
~overlapped_ptr	Destructor automatically frees the OVERLAPPED object unless released.

A special-purpose smart pointer used to wrap an application handler so that it can be passed as the LPOVERLAPPED argument to overlapped I/O functions.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/windows/overlapped_ptr.hpp

Convenience header: asio.hpp

5.228.1 windows::overlapped_ptr::complete

Post completion notification for overlapped operation. Releases ownership.

```
void complete(
    const asio::error_code & ec,
    std::size_t bytes_transferred);
```

5.228.2 windows::overlapped_ptr::get

Get the contained OVERLAPPED object.

```
OVERLAPPED * get();

const OVERLAPPED * get() const;
```

5.228.2.1 windows::overlapped_ptr::get (1 of 2 overloads)

Get the contained OVERLAPPED object.

```
OVERLAPPED * get();
```

5.228.2.2 windows::overlapped_ptr::get (2 of 2 overloads)

Get the contained OVERLAPPED object.

```
const OVERLAPPED * get() const;
```

5.228.3 windows::overlapped_ptr::overlapped_ptr

Construct an empty [windows::overlapped_ptr](#).

```
overlapped_ptr();
```

Construct an [windows::overlapped_ptr](#) to contain the specified handler.

```
template<
    typename Handler>
explicit overlapped_ptr(
    asio::io_service & io_service,
    Handler handler);
```

5.228.3.1 windows::overlapped_ptr::overlapped_ptr (1 of 2 overloads)

Construct an empty [windows::overlapped_ptr](#).

```
overlapped_ptr();
```

5.228.3.2 windows::overlapped_ptr::overlapped_ptr (2 of 2 overloads)

Construct an [windows::overlapped_ptr](#) to contain the specified handler.

```
template<
    typename Handler>
overlapped_ptr(
    asio::io_service & io_service,
    Handler handler);
```

5.228.4 windows::overlapped_ptr::release

Release ownership of the OVERLAPPED object.

```
OVERLAPPED * release();
```

5.228.5 windows::overlapped_ptr::reset

Reset to empty.

```
void reset();
```

Reset to contain the specified handler, freeing any current OVERLAPPED object.

```
template<
    typename Handler>
void reset(
    asio::io_service & io_service,
    Handler handler);
```

5.228.5.1 windows::overlapped_ptr::reset (1 of 2 overloads)

Reset to empty.

```
void reset();
```

5.228.5.2 windows::overlapped_ptr::reset (2 of 2 overloads)

Reset to contain the specified handler, freeing any current OVERLAPPED object.

```
template<
    typename Handler>
void reset(
    asio::io_service & io_service,
    Handler handler);
```

5.228.6 windows::overlapped_ptr::~overlapped_ptr

Destructor automatically frees the OVERLAPPED object unless released.

```
~overlapped_ptr();
```

5.229 windows::random_access_handle

Typedef for the typical usage of a random-access handle.

```
typedef basic_random_access_handle random_access_handle;
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_handle is always the lowest layer.
native_handle_type	The native representation of a handle.

Name	Description
native_type	(Deprecated: Use native_handle_type.) The native representation of a handle.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
async_read_some_at	Start an asynchronous read at the specified offset.
async_write_some_at	Start an asynchronous write at the specified offset.
basic_random_access_handle	Construct a basic_random_access_handle without opening it. Construct a basic_random_access_handle on an existing native handle. Move-construct a basic_random_access_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_random_access_handle from another.
read_some_at	Read some data from the handle at the specified offset.
write_some_at	Write some data to the handle at the specified offset.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.

Name	Description
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_random_access_handle` class template provides asynchronous and blocking random-access handle functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/random_access_handle.hpp`

Convenience header: `asio.hpp`

5.230 windows::random_access_handle_service

Default service implementation for a random-access handle.

```
class random_access_handle_service :
    public io_service::service
```

Types

Name	Description
implementation_type	The type of a random-access handle implementation.
native_handle_type	The native handle type.
native_type	(Deprecated: Use native_handle_type.) The native handle type.

Member Functions

Name	Description
assign	Assign an existing native handle to a random-access handle.
async_read_some_at	Start an asynchronous read at the specified offset.
async_write_some_at	Start an asynchronous write at the specified offset.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close a random-access handle implementation.
construct	Construct a new random-access handle implementation.
destroy	Destroy a random-access handle implementation.
get_io_service	Get the io_service object that owns the service.
is_open	Determine whether the handle is open.
move_assign	Move-assign from another random-access handle implementation.
move_construct	Move-construct a new random-access handle implementation.
native	(Deprecated: Use native_handle().) Get the native handle implementation.
native_handle	Get the native handle implementation.
random_access_handle_service	Construct a new random-access handle service for the specified io_service.
read_some_at	Read some data from the specified offset.
write_some_at	Write the given data at the specified offset.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/windows/random_access_handle_service.hpp

Convenience header: asio.hpp

5.230.1 windows::random_access_handle_service::assign

Assign an existing native handle to a random-access handle.

```
asio::error_code assign(
    implementation_type & impl,
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.230.2 windows::random_access_handle_service::async_read_some_at

Start an asynchronous read at the specified offset.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some_at(
    implementation_type & impl,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

5.230.3 windows::random_access_handle_service::async_write_some_at

Start an asynchronous write at the specified offset.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some_at(
    implementation_type & impl,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

5.230.4 windows::random_access_handle_service::cancel

Cancel all asynchronous operations associated with the handle.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.230.5 windows::random_access_handle_service::close

Close a random-access handle implementation.

```
asio::error_code close(
    implementation_type & impl,
    asio::error_code & ec);
```

5.230.6 windows::random_access_handle_service::construct

Construct a new random-access handle implementation.

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

5.230.7 windows::random_access_handle_service::destroy

Destroy a random-access handle implementation.

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

5.230.8 windows::random_access_handle_service::get_io_service

Inherited from io_service.

Get the **io_service** object that owns the service.

```
asio::io_service & get_io_service();
```

5.230.9 windows::random_access_handle_service::id

The unique service identifier.

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

5.230.10 windows::random_access_handle_service::implementation_type

The type of a random-access handle implementation.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asio/windows/random_access_handle_service.hpp

Convenience header: asio.hpp

5.230.11 windows::random_access_handle_service::is_open

Determine whether the handle is open.

```
bool is_open(
    const implementation_type & impl) const;
```

5.230.12 windows::random_access_handle_service::move_assign

Move-assign from another random-access handle implementation.

```
void move_assign(
    implementation_type & impl,
    random_access_handle_service & other_service,
    implementation_type & other_impl);
```

5.230.13 windows::random_access_handle_service::move_construct

Move-construct a new random-access handle implementation.

```
void move_construct(
    implementation_type & impl,
    implementation_type & other_impl);
```

5.230.14 windows::random_access_handle_service::native

(Deprecated: Use native_handle().) Get the native handle implementation.

```
native_type native(
    implementation_type & impl);
```

5.230.15 windows::random_access_handle_service::native_handle

Get the native handle implementation.

```
native_handle_type native_handle(
    implementation_type & impl);
```

5.230.16 windows::random_access_handle_service::native_handle_type

The native handle type.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/windows/random_access_handle_service.hpp

Convenience header: asio.hpp

5.230.17 windows::random_access_handle_service::native_type

(Deprecated: Use native_handle_type.) The native handle type.

```
typedef implementation_defined native_type;
```

Requirements

Header: asio/windows/random_access_handle_service.hpp

Convenience header: asio.hpp

5.230.18 windows::random_access_handle_service::random_access_handle_service

Construct a new random-access handle service for the specified **io_service**.

```
random_access_handle_service(
    asio::io_service & io_service);
```

5.230.19 windows::random_access_handle_service::read_some_at

Read some data from the specified offset.

```
template<
    typename MutableBufferSequence>
std::size_t read_some_at(
    implementation_type & impl,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.230.20 windows::random_access_handle_service::write_some_at

Write the given data at the specified offset.

```
template<
    typename ConstBufferSequence>
std::size_t write_some_at(
    implementation_type & impl,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.231 windows::stream_handle

Typedef for the typical usage of a stream-oriented handle.

```
typedef basic_stream_handle stream_handle;
```

Types

Name	Description
implementation_type	The underlying implementation type of I/O object.
lowest_layer_type	A basic_handle is always the lowest layer.
native_handle_type	The native representation of a handle.
native_type	(Deprecated: Use native_handle_type.) The native representation of a handle.
service_type	The type of the service that will be used to provide I/O operations.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.

Name	Description
async_read_some	Start an asynchronous read.
async_write_some	Start an asynchronous write.
basic_stream_handle	Construct a basic_stream_handle without opening it. Construct a basic_stream_handle on an existing native handle. Move-construct a basic_stream_handle from another.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_io_service	Get the io_service associated with the object.
is_open	Determine whether the handle is open.
lowest_layer	Get a reference to the lowest layer. Get a const reference to the lowest layer.
native	(Deprecated: Use native_handle().) Get the native handle representation.
native_handle	Get the native handle representation.
operator=	Move-assign a basic_stream_handle from another.
read_some	Read some data from the handle.
write_some	Write some data to the handle.

Protected Member Functions

Name	Description
get_implementation	Get the underlying implementation of the I/O object.
get_service	Get the service associated with the I/O object.

Protected Data Members

Name	Description
implementation	(Deprecated: Use get_implementation().) The underlying implementation of the I/O object.
service	(Deprecated: Use get_service().) The service associated with the I/O object.

The `windows::basic_stream_handle` class template provides asynchronous and blocking stream-oriented handle functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/stream_handle.hpp`

Convenience header: `asio.hpp`

5.232 windows::stream_handle_service

Default service implementation for a stream handle.

```
class stream_handle_service :  
    public io_service::service
```

Types

Name	Description
<code>implementation_type</code>	The type of a stream handle implementation.
<code>native_handle_type</code>	The native handle type.
<code>native_type</code>	(Deprecated: Use <code>native_handle_type</code> .) The native handle type.

Member Functions

Name	Description
<code>assign</code>	Assign an existing native handle to a stream handle.
<code>async_read_some</code>	Start an asynchronous read.
<code>async_write_some</code>	Start an asynchronous write.
<code>cancel</code>	Cancel all asynchronous operations associated with the handle.
<code>close</code>	Close a stream handle implementation.
<code>construct</code>	Construct a new stream handle implementation.
<code>destroy</code>	Destroy a stream handle implementation.
<code>get_io_service</code>	Get the <code>io_service</code> object that owns the service.

Name	Description
is_open	Determine whether the handle is open.
move_assign	Move-assign from another stream handle implementation.
move_construct	Move-construct a new stream handle implementation.
native	(Deprecated: Use native_handle().) Get the native handle implementation.
native_handle	Get the native handle implementation.
read_some	Read some data from the stream.
stream_handle_service	Construct a new stream handle service for the specified io_service.
write_some	Write the given data to the stream.

Data Members

Name	Description
id	The unique service identifier.

Requirements

Header: asio/windows/stream_handle_service.hpp

Convenience header: asio.hpp

5.232.1 windows::stream_handle_service::assign

Assign an existing native handle to a stream handle.

```
asio::error_code assign(
    implementation_type & impl,
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.232.2 windows::stream_handle_service::async_read_some

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
void-or-deduced async_read_some(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    ReadHandler handler);
```

5.232.3 windows::stream_handle_service::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
void-or-deduced async_write_some(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    WriteHandler handler);
```

5.232.4 windows::stream_handle_service::cancel

Cancel all asynchronous operations associated with the handle.

```
asio::error_code cancel(
    implementation_type & impl,
    asio::error_code & ec);
```

5.232.5 windows::stream_handle_service::close

Close a stream handle implementation.

```
asio::error_code close(
    implementation_type & impl,
    asio::error_code & ec);
```

5.232.6 windows::stream_handle_service::construct

Construct a new stream handle implementation.

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

5.232.7 windows::stream_handle_service::destroy

Destroy a stream handle implementation.

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

5.232.8 windows::stream_handle_service::get_io_service

Inherited from io_service.

Get the **io_service** object that owns the service.

```
asio::io_service & get_io_service();
```

5.232.9 windows::stream_handle_service::id

The unique service identifier.

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

5.232.10 windows::stream_handle_service::implementation_type

The type of a stream handle implementation.

```
typedef implementation_defined implementation_type;
```

Requirements

Header: asio/windows/stream_handle_service.hpp

Convenience header: asio.hpp

5.232.11 windows::stream_handle_service::is_open

Determine whether the handle is open.

```
bool is_open(
    const implementation_type & impl) const;
```

5.232.12 windows::stream_handle_service::move_assign

Move-assign from another stream handle implementation.

```
void move_assign(
    implementation_type & impl,
    stream_handle_service & other_service,
    implementation_type & other_impl);
```

5.232.13 windows::stream_handle_service::move_construct

Move-construct a new stream handle implementation.

```
void move_construct(
    implementation_type & impl,
    implementation_type & other_impl);
```

5.232.14 windows::stream_handle_service::native

(Deprecated: Use native_handle().) Get the native handle implementation.

```
native_type native(
    implementation_type & impl);
```

5.232.15 windows::stream_handle_service::native_handle

Get the native handle implementation.

```
native_handle_type native_handle(
    implementation_type & impl);
```

5.232.16 windows::stream_handle_service::native_handle_type

The native handle type.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/windows/stream_handle_service.hpp

Convenience header: asio.hpp

5.232.17 windows::stream_handle_service::native_type

(Deprecated: Use native_handle_type.) The native handle type.

```
typedef implementation_defined native_type;
```

Requirements

Header: asio/windows/stream_handle_service.hpp

Convenience header: asio.hpp

5.232.18 windows::stream_handle_service::read_some

Read some data from the stream.

```
template<
    typename MutableBufferSequence>
std::size_t read_some(
    implementation_type & impl,
    const MutableBufferSequence & buffers,
    asio::error_code & ec);
```

5.232.19 windows::stream_handle_service::stream_handle_service

Construct a new stream handle service for the specified [io_service](#).

```
stream_handle_service(
    asio::io_service & io_service);
```

5.232.20 windows::stream_handle_service::write_some

Write the given data to the stream.

```
template<
    typename ConstBufferSequence>
std::size_t write_some(
    implementation_type & impl,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.233 write

Write a certain amount of data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename ConstBufferSequence>
std::size_t write(
    SyncWriteStream & s,
    const ConstBufferSequence & buffers);

template<
    typename SyncWriteStream,
    typename ConstBufferSequence>
std::size_t write(
    SyncWriteStream & s,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);

template<
    typename SyncWriteStream,
    typename ConstBufferSequence,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition);

template<
    typename SyncWriteStream,
    typename ConstBufferSequence,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec);

template<
    typename SyncWriteStream,
    typename Allocator>
std::size_t write(
    SyncWriteStream & s,
    basic_streambuf<Allocator> & b);
```

```

template<
    typename SyncWriteStream,
    typename Allocator>
std::size_t write(
    SyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    asio::error_code & ec);

template<
    typename SyncWriteStream,
    typename Allocator,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition);

template<
    typename SyncWriteStream,
    typename Allocator,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    asio::error_code & ec);

```

Requirements

Header: asio/write.hpp

Convenience header: asio.hpp

5.233.1 write (1 of 8 overloads)

Write all of the supplied data to a stream before returning.

```

template<
    typename SyncWriteStream,
    typename ConstBufferSequence>
std::size_t write(
    SyncWriteStream & s,
    const ConstBufferSequence & buffers);

```

This function is used to write a certain number of bytes of data to a stream. The call will block until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's write_some function.

Parameters

s The stream to which the data is to be written. The type must support the SyncWriteStream concept.

buffers One or more buffers containing the data to be written. The sum of the buffer sizes indicates the maximum number of bytes to write to the stream.

Return Value

The number of bytes transferred.

Exceptions

asio::system_error Thrown on failure.

Example

To write a single data buffer use the **buffer** function as follows:

```
asio::write(s, asio::buffer(data, size));
```

See the **buffer** documentation for information on writing multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

Remarks

This overload is equivalent to calling:

```
asio::write(
    s, buffers,
    asio::transfer_all());
```

5.233.2 write (2 of 8 overloads)

Write all of the supplied data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename ConstBufferSequence>
std::size_t write(
    SyncWriteStream & s,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to write a certain number of bytes of data to a stream. The call will block until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's write_some function.

Parameters

- s** The stream to which the data is to be written. The type must support the SyncWriteStream concept.
- buffers** One or more buffers containing the data to be written. The sum of the buffer sizes indicates the maximum number of bytes to write to the stream.
- ec** Set to indicate what error occurred, if any.

Return Value

The number of bytes transferred.

Example

To write a single data buffer use the **buffer** function as follows:

```
asio::write(s, asio::buffer(data, size), ec);
```

See the **buffer** documentation for information on writing multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

Remarks

This overload is equivalent to calling:

```
asio::write(  
    s, buffers,  
    asio::transfer_all(), ec);
```

5.233.3 write (3 of 8 overloads)

Write a certain amount of data to a stream before returning.

```
template<  
    typename SyncWriteStream,  
    typename ConstBufferSequence,  
    typename CompletionCondition>  
std::size_t write(  
    SyncWriteStream & s,  
    const ConstBufferSequence & buffers,  
    CompletionCondition completion_condition);
```

This function is used to write a certain number of bytes of data to a stream. The call will block until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The completion_condition function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's write_some function.

Parameters

s The stream to which the data is to be written. The type must support the SyncWriteStream concept.

buffers One or more buffers containing the data to be written. The sum of the buffer sizes indicates the maximum number of bytes to write to the stream.

completion_condition The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest write_some operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the stream's write_some function.

Return Value

The number of bytes transferred.

Exceptions

asio::system_error Thrown on failure.

Example

To write a single data buffer use the **buffer** function as follows:

```
asio::write(s, asio::buffer(data, size),
            asio::transfer_at_least(32));
```

See the **buffer** documentation for information on writing multiple buffers in one go, and how to use it with arrays, boost::array or std::vector.

5.233.4 write (4 of 8 overloads)

Write a certain amount of data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename ConstBufferSequence,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec);
```

This function is used to write a certain number of bytes of data to a stream. The call will block until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The completion_condition function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's write_some function.

Parameters

- s The stream to which the data is to be written. The type must support the SyncWriteStream concept.
- buffers One or more buffers containing the data to be written. The sum of the buffer sizes indicates the maximum number of bytes to write to the stream.

completion_condition The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest write_some operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the stream's write_some function.

- ec Set to indicate what error occurred, if any.

Return Value

The number of bytes written. If an error occurs, returns the total number of bytes successfully transferred prior to the error.

5.233.5 write (5 of 8 overloads)

Write all of the supplied data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename Allocator>
std::size_t write(
    SyncWriteStream & s,
    basic_streambuf<Allocator> & b);
```

This function is used to write a certain number of bytes of data to a stream. The call will block until one of the following conditions is true:

- All of the data in the supplied **basic_streambuf** has been written.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's write_some function.

Parameters

- s The stream to which the data is to be written. The type must support the SyncWriteStream concept.
- b The **basic_streambuf** object from which data will be written.

Return Value

The number of bytes transferred.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

This overload is equivalent to calling:

```
asio::write(
    s, b,
    asio::transfer_all());
```

5.233.6 write (6 of 8 overloads)

Write all of the supplied data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename Allocator>
std::size_t write(
    SyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    asio::error_code & ec);
```

This function is used to write a certain number of bytes of data to a stream. The call will block until one of the following conditions is true:

- All of the data in the supplied `basic_streambuf` has been written.
- An error occurred.

This operation is implemented in terms of zero or more calls to the stream's `write_some` function.

Parameters

s The stream to which the data is to be written. The type must support the `SyncWriteStream` concept.

b The `basic_streambuf` object from which data will be written.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes transferred.

Remarks

This overload is equivalent to calling:

```
asio::write(
    s, b,
    asio::transfer_all(), ec);
```

5.233.7 write (7 of 8 overloads)

Write a certain amount of data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename Allocator,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition);
```

This function is used to write a certain number of bytes of data to a stream. The call will block until one of the following conditions is true:

- All of the data in the supplied **basic_streambuf** has been written.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's `write_some` function.

Parameters

s The stream to which the data is to be written. The type must support the `SyncWriteStream` concept.

b The `basic_streambuf` object from which data will be written.

completion_condition The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest write_some operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the stream's `write_some` function.

Return Value

The number of bytes transferred.

Exceptions

`asio::system_error` Thrown on failure.

5.233.8 write (8 of 8 overloads)

Write a certain amount of data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename Allocator,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    asio::error_code & ec);
```

This function is used to write a certain number of bytes of data to a stream. The call will block until one of the following conditions is true:

- All of the data in the supplied `basic_streambuf` has been written.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the stream's `write_some` function.

Parameters

s The stream to which the data is to be written. The type must support the `SyncWriteStream` concept.

b The `basic_streambuf` object from which data will be written.

completion_condition The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest write_some operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the stream's `write_some` function.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes written. If an error occurs, returns the total number of bytes successfully transferred prior to the error.

5.234 write_at

Write a certain amount of data at a specified offset before returning.

```

template<
    typename SyncRandomAccessWriteDevice,
    typename ConstBufferSequence>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers);

template<
    typename SyncRandomAccessWriteDevice,
    typename ConstBufferSequence>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);

template<
    typename SyncRandomAccessWriteDevice,
    typename ConstBufferSequence,
    typename CompletionCondition>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition);

template<
    typename SyncRandomAccessWriteDevice,
    typename ConstBufferSequence,
    typename CompletionCondition>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec);

template<
    typename SyncRandomAccessWriteDevice,
    typename Allocator>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b);

template<
    typename SyncRandomAccessWriteDevice,
    typename Allocator>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    asio::error_code & ec);

```

```

template<
    typename SyncRandomAccessWriteDevice,
    typename Allocator,
    typename CompletionCondition>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition);

template<
    typename SyncRandomAccessWriteDevice,
    typename Allocator,
    typename CompletionCondition>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    asio::error_code & ec);

```

Requirements

Header: asio/write_at.hpp

Convenience header: asio.hpp

5.234.1 write_at (1 of 8 overloads)

Write all of the supplied data at the specified offset before returning.

```

template<
    typename SyncRandomAccessWriteDevice,
    typename ConstBufferSequence>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers);

```

This function is used to write a certain number of bytes of data to a random access device at a specified offset. The call will block until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the device's write_some_at function.

Parameters

d The device to which the data is to be written. The type must support the SyncRandomAccessWriteDevice concept.

offset The offset at which the data will be written.

buffers One or more buffers containing the data to be written. The sum of the buffer sizes indicates the maximum number of bytes to write to the device.

Return Value

The number of bytes transferred.

Exceptions

`asio::system_error` Thrown on failure.

Example

To write a single data buffer use the `buffer` function as follows:

```
asio::write_at(d, 42, asio::buffer(data, size));
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

Remarks

This overload is equivalent to calling:

```
asio::write_at(
    d, offset, buffers,
    asio::transfer_all());
```

5.234.2 `write_at` (2 of 8 overloads)

Write all of the supplied data at the specified offset before returning.

```
template<
    typename SyncRandomAccessWriteDevice,
    typename ConstBufferSequence>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

This function is used to write a certain number of bytes of data to a random access device at a specified offset. The call will block until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- An error occurred.

This operation is implemented in terms of zero or more calls to the device's `write_some_at` function.

Parameters

d The device to which the data is to be written. The type must support the `SyncRandomAccessWriteDevice` concept.

offset The offset at which the data will be written.

buffers One or more buffers containing the data to be written. The sum of the buffer sizes indicates the maximum number of bytes to write to the device.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes transferred.

Example

To write a single data buffer use the **buffer** function as follows:

```
asio::write_at(d, 42,
    asio::buffer(data, size), ec);
```

See the **buffer** documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

Remarks

This overload is equivalent to calling:

```
asio::write_at(
    d, offset, buffers,
    asio::transfer_all(), ec);
```

5.234.3 write_at (3 of 8 overloads)

Write a certain amount of data at a specified offset before returning.

```
template<
    typename SyncRandomAccessWriteDevice,
    typename ConstBufferSequence,
    typename CompletionCondition>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition);
```

This function is used to write a certain number of bytes of data to a random access device at a specified offset. The call will block until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the device's `write_some_at` function.

Parameters

d The device to which the data is to be written. The type must support the `SyncRandomAccessWriteDevice` concept.

offset The offset at which the data will be written.

buffers One or more buffers containing the data to be written. The sum of the buffer sizes indicates the maximum number of bytes to write to the device.

completion_condition The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```

std::size_t completion_condition(
    // Result of latest write_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);

```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the device's `write_some_at` function.

Return Value

The number of bytes transferred.

Exceptions

asio::system_error Thrown on failure.

Example

To write a single data buffer use the `buffer` function as follows:

```
asio::write_at(d, 42, asio::buffer(data, size),
               asio::transfer_at_least(32));
```

See the `buffer` documentation for information on writing multiple buffers in one go, and how to use it with arrays, `boost::array` or `std::vector`.

5.234.4 write_at (4 of 8 overloads)

Write a certain amount of data at a specified offset before returning.

```

template<
    typename SyncRandomAccessWriteDevice,
    typename ConstBufferSequence,
    typename CompletionCondition>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec);

```

This function is used to write a certain number of bytes of data to a random access device at a specified offset. The call will block until one of the following conditions is true:

- All of the data in the supplied buffers has been written. That is, the bytes transferred is equal to the sum of the buffer sizes.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the device's `write_some_at` function.

Parameters

d The device to which the data is to be written. The type must support the SyncRandomAccessWriteDevice concept.

offset The offset at which the data will be written.

buffers One or more buffers containing the data to be written. The sum of the buffer sizes indicates the maximum number of bytes to write to the device.

completion_condition The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest write_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the device's write_some_at function.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes written. If an error occurs, returns the total number of bytes successfully transferred prior to the error.

5.234.5 write_at (5 of 8 overloads)

Write all of the supplied data at the specified offset before returning.

```
template<
    typename SyncRandomAccessWriteDevice,
    typename Allocator>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b);
```

This function is used to write a certain number of bytes of data to a random access device at a specified offset. The call will block until one of the following conditions is true:

- All of the data in the supplied `basic_streambuf` has been written.
- An error occurred.

This operation is implemented in terms of zero or more calls to the device's write_some_at function.

Parameters

d The device to which the data is to be written. The type must support the SyncRandomAccessWriteDevice concept.

offset The offset at which the data will be written.

b The `basic_streambuf` object from which data will be written.

Return Value

The number of bytes transferred.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

This overload is equivalent to calling:

```
asio::write_at(
    d, 42, b,
    asio::transfer_all());
```

5.234.6 write_at (6 of 8 overloads)

Write all of the supplied data at the specified offset before returning.

```
template<
    typename SyncRandomAccessWriteDevice,
    typename Allocator>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf<Allocator> & b,
    asio::error_code & ec);
```

This function is used to write a certain number of bytes of data to a random access device at a specified offset. The call will block until one of the following conditions is true:

- All of the data in the supplied `basic_streambuf` has been written.
- An error occurred.

This operation is implemented in terms of zero or more calls to the device's `write_some_at` function.

Parameters

d The device to which the data is to be written. The type must support the `SyncRandomAccessWriteDevice` concept.

offset The offset at which the data will be written.

b The `basic_streambuf` object from which data will be written.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes transferred.

Remarks

This overload is equivalent to calling:

```
asio::write_at(
    d, 42, b,
    asio::transfer_all(), ec);
```

5.234.7 write_at (7 of 8 overloads)

Write a certain amount of data at a specified offset before returning.

```
template<
    typename SyncRandomAccessWriteDevice,
    typename Allocator,
    typename CompletionCondition>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition);
```

This function is used to write a certain number of bytes of data to a random access device at a specified offset. The call will block until one of the following conditions is true:

- All of the data in the supplied `basic_streambuf` has been written.
- The `completion_condition` function object returns 0.

This operation is implemented in terms of zero or more calls to the device's `write_some_at` function.

Parameters

d The device to which the data is to be written. The type must support the `SyncRandomAccessWriteDevice` concept.

offset The offset at which the data will be written.

b The `basic_streambuf` object from which data will be written.

completion_condition The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest write_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the device's `write_some_at` function.

Return Value

The number of bytes transferred.

Exceptions

asio::system_error Thrown on failure.

5.234.8 write_at (8 of 8 overloads)

Write a certain amount of data at a specified offset before returning.

```
template<
    typename SyncRandomAccessWriteDevice,
    typename Allocator,
    typename CompletionCondition>
std::size_t write_at(
    SyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    asio::error_code & ec);
```

This function is used to write a certain number of bytes of data to a random access device at a specified offset. The call will block until one of the following conditions is true:

- All of the data in the supplied **basic_streambuf** has been written.
- The **completion_condition** function object returns 0.

This operation is implemented in terms of zero or more calls to the device's **write_some_at** function.

Parameters

d The device to which the data is to be written. The type must support the **SyncRandomAccessWriteDevice** concept.

offset The offset at which the data will be written.

b The **basic_streambuf** object from which data will be written.

completion_condition The function object to be called to determine whether the write operation is complete. The signature of the function object must be:

```
std::size_t completion_condition(
    // Result of latest write_some_at operation.
    const asio::error_code& error,
    // Number of bytes transferred so far.
    std::size_t bytes_transferred
);
```

A return value of 0 indicates that the write operation is complete. A non-zero return value indicates the maximum number of bytes to be written on the next call to the device's **write_some_at** function.

ec Set to indicate what error occurred, if any.

Return Value

The number of bytes written. If an error occurs, returns the total number of bytes successfully transferred prior to the error.

5.235 yield_context

Context object that represents the currently executing coroutine.

```
typedef basic_yield_context< unspecified > yield_context;
```

Types

Name	Description
callee_type	The coroutine callee type, used by the implementation.
caller_type	The coroutine caller type, used by the implementation.

Member Functions

Name	Description
basic_yield_context	Construct a yield context to represent the specified coroutine.
operator[]	Return a yield context that sets the specified error_code.

The `basic_yield_context` class is used to represent the currently executing stackful coroutine. A `basic_yield_context` may be passed as a handler to an asynchronous operation. For example:

```
template <typename Handler>
void my_coroutine(basic_yield_context<Handler> yield)
{
    ...
    std::size_t n = my_socket.async_read_some(buffer, yield);
    ...
}
```

The initiating function (`async_read_some` in the above example) suspends the current coroutine. The coroutine is resumed when the asynchronous operation completes, and the result of the operation is returned.

Requirements

Header: `asio/spawn.hpp`

Convenience header: None

6 Revision History

Asio 1.10.6

- Ensured errors generated by Windows' `ConnectEx` function are mapped to their portable equivalents.
- Added new macro `(BOOST_) ASIO_DISABLE_CONNECTEX` to allow use of `ConnectEx` to be explicitly disabled.
- Fixed a race condition in `windows::object_handle` when there are pending wait operations on destruction.
- Fixed IPv6 address parsing on FreeBSD, where a trailing scope ID would cause conversion to fail with `EINVAL`.
- Worked around shared library visibility issues by ensuring Asio types use default visibility.
- Changed the SSL wrapper to call the password callback when loading an in-memory key.
- Fixed false SSL error reports by ensuring that the SSL error queue is cleared prior to each operation.
- Fixed an `ssl::stream` bug that may result in spurious 'short read' errors.
- Removed a redundant null pointer check in the SSL engine.
- Added options for disabling TLS v1.1 and v1.2.
- Removed use of deprecated OpenSSL function `ERR_remove_state`.
- Fixed detection of various C++11 features with Clang.
- Fixed detection of C++11 `std::addressof` with g++.
- Changed multicast test to treat certain `join_group` failures as non-fatal.
- Decoupled Asio unit tests from Boost.Test.
- Changed the tutorial to use `std::endl` to ensure output is flushed.
- Fixed an unsigned integer overflow reported by Clang's integer sanitizer.
- Added support for move-only return types when using a `yield_context` object with asynchronous operations.
- Changed `yield_context` to allow reentrant calls to the completion handler from an initiating function.
- Updated detection of Windows Runtime to work with latest Windows SDK.

Asio 1.10.5

- Fixed the `kqueue` reactor so that it works on FreeBSD.
- Fixed an issue in the `kqueue` reactor which resulted in spinning when using serial ports on Mac OS.
- Fixed `kqueue` reactor support for read-only file descriptors.
- Fixed a compile error when using the `/dev/poll` reactor.
- Changed the Windows backend to use `WSASocketW`, as `WSASocketA` has been deprecated.
- Fixed some warnings reported by Visual C++ 2013.
- Fixed integer type used in the WinRT version of the byte-order conversion functions.
- Changed documentation to indicate that `use_future` and `spawn()` are not made available when including the `asio.hpp` convenience header.
- Explicitly marked `asio::strand` as deprecated. Use `asio::io_service::strand` instead.

Asio 1.10.4

- Stopped using certain Winsock functions that are marked as deprecated in the latest Visual C++ and Windows SDK.
- Fixed a shadow variable warning on Windows.
- Fixed a regression in the kqueue backend that was introduced in Asio 1.10.2.
- Added a workaround for building the unit tests with `gcc` on AIX.

Asio 1.10.3

- Worked around a `gcc` problem to do with anonymous enums.
- Reverted the Windows HANDLE backend change to ignore `ERROR_MORE_DATA`. Instead, the error will be propagated as with any other (i.e. in an `error_code` or thrown as a `system_error`), and the number of bytes transferred will be returned. For code that needs to handle partial messages, the `error_code` overload should be used.
- Fixed an off-by-one error in the `signal_set` implementation's signal number check.
- Changed the Windows IOCP backend to not assume that `SO_UPDATE_CONNECT_CONTEXT` is defined.
- Fixed a Windows-specific issue, introduced in Asio 1.10.2, by using `VerifyVersionInfo` rather than `GetVersionEx`, as `GetVersionEx` has been deprecated.
- Changed to use SSE2 intrinsics rather than inline assembly, to allow the Cray compiler to work.

Asio 1.10.2

- Fixed `asio::spawn()` to work correctly with new Boost.Coroutine interface.
- Ensured that incomplete `asio::spawn()` coroutines are correctly unwound when cleaned up by the `io_service` destructor.
- Fixed delegation of continuation hook for handlers produced by `io_service::wrap()` and `strand::wrap()`.
- Changed the Windows I/O completion port backend to use `ConnectEx`, if available, for connection-oriented IP sockets.
- Changed the `io_service` backend for non-Windows (and non-IOCP Windows) platforms to use a single condition variable per `io_service` instance. This addresses a potential race condition when `run_one()` is used from multiple threads.
- Prevented integer overflow when computing timeouts based on some `boost::chrono` and `std::chrono` clocks.
- Made further changes to `EV_CLEAR` handling in the kqueue backend, to address other cases where the `close()` system call may hang on Mac OS X.
- Fixed infinite recursion in implementation of `resolver_query_base::flags::operator~`.
- Made the `select` reactor more efficient on Windows for large numbers of sockets.
- Fixed a Windows-specific type-aliasing issue reported by `gcc`.
- Prevented execution of compile-time-only buffer test to avoid triggering an address sanitiser warning.
- Disabled the `GetQueuedCompletionStatus` timeout workaround on recent versions of Windows.
- Changed implementation for Windows Runtime to use `FormatMessageW` rather than `FormatMessageA`, as the Windows store does not permit the latter.
- Added support for string-based scope IDs when using link-local multicast addresses.
- Changed IPv6 multicast group join to use the address's scope ID as the interface, if an interface is not explicitly specified.
- Fixed multicast test failure on Mac OS X and the BSDs by using a link-local multicast address.
- Various minor documentation improvements.

Asio 1.10.1

- Implemented a limited port to Windows Runtime. This support requires that the language extensions be enabled. Due to the restricted facilities exposed by the Windows Runtime API, the port also comes with the following caveats:
 - The core facilities such as the `io_service`, `strand`, `buffers`, composed operations, timers, etc., should all work as normal.
 - For sockets, only client-side TCP is supported.
 - Explicit binding of a client-side TCP socket is not supported.
 - The `cancel()` function is not supported for sockets. Asynchronous operations may only be cancelled by closing the socket.
 - Operations that use `null_buffers` are not supported.
 - Only `tcp::no_delay` and `socket_base::keep_alive` options are supported.
 - Resolvers do not support service names, only numbers. I.e. you must use "80" rather than "http".
 - Most resolver query flags have no effect.
- Extended the ability to use Asio without Boost to include Microsoft Visual Studio 2012. When using a C++11 compiler, most of Asio may now be used without a dependency on Boost header files or libraries. To use Asio in this way, define `ASIO_STANDALONE` on your compiler command line or as part of the project options. This standalone configuration has been tested for the following platforms and compilers:
 - Microsoft Visual Studio 2012
 - Linux with g++ 4.7 or 4.8 (requires `-std=c++11`)
 - Mac OS X with clang++ / Xcode 4.6 (requires `-std=c++11 -stdlib=libc++`)
- Fixed a regression (introduced in 1.10.0) where, on some platforms, errors from `async_connect` were not correctly propagated through to the completion handler.
- Fixed a Windows-specific regression (introduced in 1.10.0) that occurs when multiple threads are running an `io_service`. When the bug occurs, the result of an asynchronous operation (error and bytes transferred) is incorrectly discarded and zero values used instead. For TCP sockets this results in spurious end-of-file notifications.
- Fixed a bug in handler tracking, where it was not correctly printing out some handler IDs.
- Fixed the comparison used to test for successful synchronous accept operations so that it works correctly with unsigned socket descriptors.
- Ensured the signal number is correctly passed to the completion handler when starting an `async_wait` on a signal that is already raised.
- Suppressed a g++ 4.8+ warning about unused typedefs.
- Enabled the move optimisation for handlers that use the default invocation hook.
- Clarified that programs must not issue overlapping `async_write_at` operations.
- Changed the Windows HANDLE backend to treat `ERROR_MORE_DATA` as a non-fatal error when returned by `GetOverlappedResult` for a synchronous read.
- Visual C++ language extensions use `generic` as a keyword. Added a workaround that renames the namespace to `cpp_gen generic` when those language extensions are in effect.
- Fixed some asynchronous operations that missed out on getting `async_result` support in 1.10.0. In particular, the buffered stream templates have been updated so that they adhere to current handler patterns.
- Enabled move support for Microsoft Visual Studio 2012.
- Added `use_future` support for Microsoft Visual Studio 2012.
- Removed a use of `std::min` in the Windows IOCP backend to avoid a dependency on the `<algorithm>` header.

- Eliminated some unnecessary handler copies.
- Fixed support for older versions of OpenSSL that do not provide the `SSL_CTX_clear_options` function.
- Fixed various minor and cosmetic issues in code and documentation.

Asio 1.10.0

- Added new traits classes, `handler_type` and `async_result`, that allow the customisation of the return type of an initiating function.
- Added the `asio::spawn()` function, a high-level wrapper for running stackful coroutines, based on the Boost.Coroutine library. The `spawn()` function enables programs to implement asynchronous logic in a synchronous manner. For example: `size_t n =my_socket.async_read_some(my_buffer, yield);`. For further information, see [Stackful Coroutines](#).
- Added the `asio::use_future` special value, which provides first-class support for returning a C++11 `std::future` from an asynchronous operation's initiating function. For example: `future<size_t> =my_socket.async_read_some(my_buffer, asio::use_future);`. For further information, see [C++ 2011 Support - Futures](#).
- Promoted the stackless coroutine class and macros to be part of Asio's documented interface, rather than part of the HTTP server 4 example. For further information, see [Stackless Coroutines](#).
- Added a new handler hook called `asio_handler_is_continuation`. Asynchronous operations may represent a continuation of the asynchronous control flow associated with the current executing handler. The `asio_handler_is_continuation` hook can be customised to return `true` if this is the case, and Asio's implementation can use this knowledge to optimise scheduling of the new handler. To cover common cases, Asio customises the hook for strands, `spawn()` and composed asynchronous operations.
- Added four new generic protocol classes, `generic::datagram_protocol`, `generic::raw_protocol`, `generic::seq_packet_protocol` and `generic::stream_protocol`, which implement the `Protocol` type requirements, but allow the user to specify the address family (e.g. `AF_INET`) and protocol type (e.g. `IPPROTO_TCP`) at runtime. For further information, see [Support for Other Protocols](#).
- Added C++11 move constructors that allow the conversion of a socket (or acceptor) into a more generic type. For example, an `ip::tcp::socket` can be converted into a `generic::stream_protocol::socket` via move construction. For further information, see [Support for Other Protocols](#).
- Extended the `basic_socket_acceptor<>`'s `accept()` and `async_accept()` functions to allow a new connection to be accepted directly into a socket of a more generic type. For example, an `ip::tcp::acceptor` can be used to accept into a `generic::stream_protocol::socket` object. For further information, see [Support for Other Protocols](#).
- Moved existing examples into a C++03-specific directory, and added a new directory for C++11-specific examples. A limited subset of the C++03 examples have been converted to their C++11 equivalents.
- Add the ability to use Asio without Boost, for a limited set of platforms. When using a C++11 compiler, most of Asio may now be used without a dependency on Boost header files or libraries. To use Asio in this way, define `ASIO_STANDALONE` on your compiler command line or as part of the project options. This standalone configuration has currently been tested for the following platforms and compilers:
 - Linux with g++ 4.7 (requires `-std=c++11`)
 - Mac OS X with clang++ / Xcode 4.6 (requires `-std=c++11 -stdlib=libc++`)
- Various SSL enhancements. Thanks go to Nick Jones, on whose work these changes are based.
 - Added support for SSL handshakes with re-use of data already read from the wire. New overloads of the `ssl::stream<>` class's `handshake()` and `async_handshake()` functions have been added. These accept a `ConstBufferSequence` to be used as initial input to the `ssl` engine for the handshake procedure.
 - Added support for creation of TLSv1.1 and TLSv1.2 `ssl::context` objects.
 - Added a `set_verify_depth()` function to the `ssl::context` and `ssl::stream<>` classes.

- Added the ability to load SSL certificate and key data from memory buffers. New functions, `add_certificate_authority()`, `use_certificate()`, `use_certificate_chain()`, `use_private_key()`, `use_rsa_private_key()` and `use_tmp_dh()`, have been added to the `ssl::context` class.
- Changed `ssl::context` to automatically disable SSL compression by default. To enable, use the new `ssl::context::clear_options()` function, as in `my_context.clear_options(ssl::context::no_compression)`.
- Fixed a potential deadlock in `signal_set` implementation.
- Fixed an error in acceptor example in documentation.
- Fixed copy-paste errors in waitable timer documentation.
- Added assertions to satisfy some code analysis tools.
- Fixed a malformed `#warning` directive.
- Fixed a potential data race in the Linux `epoll` implementation.
- Fixed a Windows-specific bug, where certain operations might generate an `error_code` with an invalid (i.e. `NULL`) `error_category`.
- Fixed `basic_waitable_timer`'s underlying implementation so that it can handle any `time_point` value without overflowing the intermediate duration objects.
- Fixed a problem with lost thread wakeups that can occur when making concurrent calls to `run()` and `poll()` on the same `io_service` object.
- Fixed implementation of asynchronous connect operation so that it can cope with spurious readiness notifications from the reactor.
- Fixed a memory leak in the `ssl::rfc2818_verification` class.
- Added a mechanism for disabling automatic Winsock initialisation. See the header file `asio/detail/winsock_init.hpp` for details.

Asio 1.8.3

- Fixed some 64-to-32-bit conversion warnings.
- Fixed various small errors in documentation and comments.
- Fixed an error in the example embedded in `basic_socket::get_option`'s documentation.
- Changed to use `long` rather than `int` for `SSL_CTX` options, to match OpenSSL.
- Changed to use `_snprintf` to address a compile error due to the changed `swprintf` signature in recent versions of MinGW.
- Fixed a deadlock that can occur on Windows when shutting down a pool of `io_service` threads due to running out of work.
- Changed UNIX domain socket example to treat errors from `accept` as non-fatal.
- Added a small block recycling optimisation to improve default memory allocation behaviour.

Asio 1.8.2

- Fixed an incompatibility between `ip::tcp::iostream` and C++11.
- Decorated GCC attribute names with underscores to prevent interaction with user-defined macros.
- Added missing `#include <cctype>`, needed for some versions of MinGW.
- Changed to use `gcc`'s atomic builtins on ARM CPUs, when available.
- Changed strand destruction to be a no-op, to allow strand objects to be destroyed after their associated `io_service` has been destroyed.
- Added support for some newer versions of glibc which provide the `epoll_create1()` function but always fail with ENOSYS.
- Changed the SSL implementation to throw an exception if SSL engine initialisation fails.
- Fixed another regression in `buffered_write_stream`.
- Implemented various minor performance improvements, primarily targeted at Linux x86 and x86-64 platforms.

Asio 1.8.1

- Changed the `epoll_reactor` backend to do lazy registration for EPOLLOUT events.
- Fixed the `epoll_reactor` handling of out-of-band data, which was broken by an incomplete fix in the last release.
- Changed Asio's SSL wrapper to respect OpenSSL's `OPENSSL_NO_ENGINE` feature test `#define`.
- Fixed `windows::object_handle` so that it works with Windows compilers that support C++11 move semantics (such as g++).
- Improved the performance of strand rescheduling.
- Added support for g++ 4.7 when compiling in C++11 mode.
- Fixed a problem where `signal_set` handlers were not being delivered when the `io_service` was constructed with a `concurrency_hint` of 1.

Asio 1.8.0

- Added a new class template `basic_waitable_timer` based around the C++11 clock type requirements. It may be used with the clocks from the C++11 `<chrono>` library facility or, if those are not available, Boost.Chrono. The typedefs `high_resolution_timer`, `steady_timer` and `system_timer` may be used to create timer objects for the standard clock types.
- Added a new `windows::object_handle` class for performing waits on Windows kernel objects. Thanks go to Boris Schaeling for contributing substantially to the development of this feature.
- On Linux, `connect()` can return EAGAIN in certain circumstances. Remapped this to another error so that it doesn't look like a non-blocking operation.
- Fixed a compile error on NetBSD.
- Fixed deadlock on Mac OS X.
- Fixed a regression in `buffered_write_stream`.
- Fixed a non-paged pool "leak" on Windows when an `io_service` is repeatedly run without anything to do.
- Reverted earlier change to allow some speculative operations to be performed without holding the lock, as it introduced a race condition in some multithreaded scenarios.
- Fixed a bug where the second buffer in an array of two buffers may be ignored if the first buffer is empty.

Asio 1.6.1

- Implemented various performance improvements, including:
 - Using thread-local operation queues in single-threaded use cases (i.e. when `concurrency_hint` is 1) to eliminate a lock/unlock pair.
 - Allowing some `epoll_reactor` speculative operations to be performed without holding the lock.
 - Improving locality of reference by performing an `epoll_reactor`'s I/O operation immediately before the corresponding handler is called. This also improves scalability across CPUs when multiple threads are running the `io_service`.
 - Specialising asynchronous read and write operations for buffer sequences that are arrays (`boost::array` or `std::array`) of exactly two buffers.
- Fixed a compile error in the regex overload of `async_read_until`.
- Fixed a Windows-specific compile error by explicitly specifying the `signal()` function from the global namespace.
- Changed the `deadline_timer` implementation so that it does not read the clock unless the timer heap is non-empty.
- Changed the SSL stream's buffers' sizes so that they are large enough to hold a complete TLS record.
- Fixed the behaviour of the synchronous `null_buffers` operations so that they obey the user's non-blocking setting.
- Changed to set the size of the select `fd_set` at runtime when using Windows.
- Disabled an MSVC warning due to `const` qualifier being applied to function type.
- Fixed a crash that occurs when using the Intel C++ compiler.
- Changed the initialisation of the OpenSSL library so that it supports all available algorithms.
- Fixed the SSL error mapping used when the session is gracefully shut down.
- Added some latency test programs.
- Clarified that a read operation ends when the buffer is full.
- Fixed an exception safety issue in `epoll_reactor` initialisation.
- Made the number of strand implementations configurable by defining `(BOOST_)ASIO_STRAND_IMPLEMENTATIONS` to the desired number.
- Added support for a new `(BOOST_)ASIO_ENABLE_SEQUENTIAL_STRAND_ALLOCATION` flag which switches the allocation of strand implementations to use a round-robin approach rather than hashing.
- Fixed potential strand starvation issue that can occur when `strand.post()` is used.

Asio 1.6.0

- Improved support for C++0x move construction to further reduce copying of handler objects. In certain designs it is possible to eliminate virtually all copies. Move support is now enabled when compiling in `-std=c++0x` mode on g++ 4.5 or higher.
- Added build support for platforms that don't provide either of `signal()` or `sigaction()`.
- Changed to use C++0x variadic templates when they are available, rather than generating function overloads using the Boost.Preprocessor library.
- Ensured the value of `errno` is preserved across the implementation's signal handler.
- On Windows, ensured the count of outstanding work is decremented for abandoned operations (i.e. operations that are being cleaned up within the `io_service` destructor).
- Fixed behaviour of zero-length reads and writes in the new SSL implementation.

- Added support for building with OpenSSL 1.0 when `OPENSSL_NO_SSL2` is defined.
- Changed most examples to treat a failure by an accept operation as non-fatal.
- Fixed an error in the `tick_count_timer` example by making the duration type signed. Previously, a wait on an already-passed deadline would not return for a very long time.

Asio 1.5.3

- Added a new, completely rewritten SSL implementation. The new implementation compiles faster, shows substantially improved performance, and supports custom memory allocation and handler invocation. It includes new API features such as certificate verification callbacks and has improved error reporting. The new implementation is source-compatible with the old for most uses. However, if necessary, the old implementation may still be used by defining `(BOOST_) ASIO_ENABLE_OLD_SSL`.
- Added new `asio::buffer()` overloads for `std::array`, when available. The support is automatically enabled when compiling in `-std=c++0x` mode on g++ 4.3 or higher, or when using MSVC 10. The support may be explicitly enabled by defining `(BOOST_) ASIO_HAS_STD_ARRAY`, or disabled by defining `(BOOST_) ASIO_DISABLE_STD_ARRAY`.
- Changed to use the C++0x standard library templates `array`, `shared_ptr`, `weak_ptr` and `atomic` when they are available, rather than the Boost equivalents.
- Support for `std::error_code` and `std::system_error` is no longer enabled by default for g++ 4.5, as that compiler's standard library does not implement `std::system_error::what()` correctly.

Asio 1.5.2

- Added support for C++0x move construction and assignment to sockets, serial ports, POSIX descriptors and Windows handles.
- Added support for the `fork()` system call. Programs that use `fork()` must call `io_service.notify_fork()` at the appropriate times. Two new examples have been added showing how to use this feature.
- Cleaned up the handling of errors reported by the `close()` system call. In particular, assume that most operating systems won't have `close()` fail with `EWOULDBLOCK`, but if it does then set the blocking mode and restart the call. If any other error occurs, assume the descriptor is closed.
- The kqueue flag `EV_ONESHOT` seems to cause problems on some versions of Mac OS X, with the `io_service` destructor getting stuck inside the `close()` system call. Changed the kqueue backend to use `EV_CLEAR` instead.
- Changed exception reporting to include the function name in exception `what()` messages.
- Fixed an insufficient initialisers warning with MinGW.
- Changed the `shutdown_service()` member functions to be private.
- Added archetypes for testing socket option functions.
- Added a missing lock in `signal_set_service::cancel()`.
- Fixed a copy/paste error in `SignalHandler` example.
- Added the inclusion of the signal header to `signal_set_service.hpp` so that constants like `NSIG` may be used.
- Changed the `signal_set_service` implementation so that it doesn't assume that `SIGRTMAX` is a compile-time constant.
- Changed the Boost.Asio examples so that they don't use Boost.Thread's convenience header. Use the header file that is specifically for the `boost::thread` class instead.

Asio 1.5.1

- Added support for signal handling, using a new class called `signal_set`. Programs may add one or more signals to the set, and then perform an `async_wait()` operation. The specified handler will be called when one of the signals occurs. The same signal number may be registered with multiple `signal_set` objects, however the signal number must be used only with Asio.
- Added handler tracking, a new debugging aid. When enabled by defining `(BOOST_) ASIO_ENABLE_HANDLER_TRACKING`, Asio writes debugging output to the standard error stream. The output records asynchronous operations and the relationships between their handlers. It may be post-processed using the included `handlerviz.pl` tool to create a visual representation of the handlers (requires GraphViz).
- Fixed a bug in `asio::streambuf` where the `consume()` function did not always update the internal buffer pointers correctly. The problem may occur when the `asio::streambuf` is filled with data using the standard C++ member functions such as `sputn()`. (Note: the problem does not manifest when the `streambuf` is populated by the Asio free functions `read()`, `async_read()`, `read_until()` or `async_read_until()`.)
- Fixed a bug on kqueue-based platforms, where reactor read operations that return false from their `perform()` function are not correctly re-registered with kqueue.
- Support for `std::error_code` and `std::system_error` is no longer enabled by default for MSVC10, as that compiler's standard library does not implement `std::system_error::what()` correctly.
- Modified the `buffers_iterator<>` and `ip::basic_resolver_iterator` classes so that the `value_type` typedefs are non-const byte types.

Asio 1.5.0

- Added support for timeouts on socket iostreams, such as `ip::tcp::iostream`. A timeout is set by calling `expires_at()` or `expires_from_now()` to establish a deadline. Any socket operations which occur past the deadline will put the iostream into a bad state.
- Added a new `error()` member function to socket iostreams, for retrieving the error code from the most recent system call.
- Added a new `basic_deadline_timer::cancel_one()` function. This function lets you cancel a single waiting handler on a timer. Handlers are cancelled in FIFO order.
- Added a new `transfer_exactly()` completion condition. This can be used to send or receive a specified number of bytes even if the total size of the buffer (or buffer sequence) is larger.
- Added new free functions `connect()` and `async_connect()`. These operations try each endpoint in a list until the socket is successfully connected.
- Extended the `buffer_size()` function so that it works for buffer sequences in addition to individual buffers.
- Added a new `buffer_copy()` function that can be used to copy the raw bytes between individual buffers and buffer sequences.
- Added new non-throwing overloads of `read()`, `read_at()`, `write()` and `write_at()` that do not require a completion condition.
- Added friendlier compiler errors for when a completion handler does not meet the necessary type requirements. When C++0x is available (currently supported for g++ 4.5 or later, and MSVC 10), `static_assert` is also used to generate an informative error message. This checking may be disabled by defining `(BOOST_) ASIO_DISABLE_HANDLER_TYPE_REQUIREMENTS`.
- Added support for using `std::error_code` and `std::system_error`, when available. The support is automatically enabled when compiling in `-std=c++0x` mode on g++ 4.5 or higher, or when using MSVC 10. The support may be explicitly enabled by defining `ASIO_HAS_STD_SYSTEM_ERROR`, or disabled by defining `ASIO_DISABLE_STD_SYSTEM_ERROR`. (Available in non-Boost version of Asio only.)

- Made the `is_loopback()`, `is_unspecified()` and `is_multicast()` functions consistently available across the `ip::address`, `ip::address_v4` and `ip::address_v6` classes.
- Added new `non_blocking()` functions for managing the non-blocking behaviour of a socket or descriptor. The `io_control()` commands named `non_blocking_io` are now deprecated in favour of these new functions.
- Added new `native_non_blocking()` functions for managing the non-blocking mode of the underlying socket or descriptor. These functions are intended to allow the encapsulation of arbitrary non-blocking system calls as asynchronous operations, in a way that is transparent to the user of the socket object. The functions have no effect on the behaviour of the synchronous operations of the socket or descriptor.
- Added the `io_control()` member function for socket acceptors.
- For consistency with the C++0x standard library, deprecated the `native_type` typedefs in favour of `native_handle_t`, and the `native()` member functions in favour of `native_handle()`.
- Added a `release()` member function to posix descriptors. This function releases ownership of the underlying native descriptor to the caller.
- Added support for sequenced packet sockets (`SOCK_SEQPACKET`).
- Added a new `io_service::stopped()` function that can be used to determine whether the `io_service` has stopped (i.e. a `reset()` call is needed prior to any further calls to `run()`, `run_one()`, `poll()` or `poll_one()`).
- Reduced the copying of handler function objects.
- Added support for C++0x move construction to further reduce copying of handler objects. Move support is enabled when compiling in `-std=c++0x` mode on g++ 4.5 or higher, or when using MSVC10.
- Removed the dependency on OS-provided macros for the well-known IPv4 and IPv6 addresses. This should eliminate the annoying "missing braces around initializer" warnings.
- Reduced the size of `ip::basic_endpoint<>` objects (such as `ip::tcp::endpoint` and `ip::udp::endpoint`).
- Changed the reactor backends to assume that any descriptors or sockets added using `assign()` may have been `dup()`-ed, and so require explicit deregistration from the reactor.
- Changed the SSL error category to return error strings from the OpenSSL library.
- Changed the separate compilation support such that, to use Asio's SSL capabilities, you should also include `'asio/ssl/impl/src.hpp'` in one source file in your program.
- Removed the deprecated member functions named `io_service()`. The `get_io_service()` member functions should be used instead.
- Removed the deprecated typedefs `resolver_query` and `resolver_iterator` from the `ip::tcp`, `ip::udp` and `ip::icmp` classes.
- Fixed a compile error on some versions of g++ due to anonymous enums.
- Added an explicit cast to the `FIONBIO` constant to int to suppress a compiler warning on some platforms.
- Fixed warnings reported by g++'s `-Wshadow` compiler option.

Asio 1.4.8

- Fixed an integer overflow problem that occurs when `ip::address_v4::broadcast()` is used on 64-bit platforms.
- Fixed a problem on older Linux kernels (where epoll is used without timerfd support) that prevents timely delivery of `deadline_timer` handlers, after the program has been running for some time.

Asio 1.4.7

- Fixed a problem on kqueue-based platforms where a `deadline_timer` may never fire if the `io_service` is running in a background thread.
- Fixed a const-correctness issue that prevented valid uses of `has_service<>` from compiling.
- Fixed MinGW cross-compilation.
- Removed dependency on deprecated Boost.System functions (Boost.Asio only).
- Ensured `close()`/`closesocket()` failures are correctly propagated.
- Added a check for errors returned by `InitializeCriticalSectionAndSpinCount`.
- Added support for hardware flow control on QNX.
- Always use `pselect()` on HP-UX, if it is available.
- Ensured handler arguments are passed as lvalues.
- Fixed Windows build when thread support is disabled.
- Fixed a Windows-specific problem where `deadline_timer` objects with expiry times set more than 5 minutes in the future may never expire.
- Fixed the resolver backend on BSD platforms so that an empty service name resolves to port number 0, as per the documentation.
- Fixed read operations so that they do not accept buffer sequences of type `const_buffers_1`.
- Redefined `Protocol` and `id` to avoid clashing with Objective-C++ keywords.
- Fixed a `vector` reallocation performance issue that can occur when there are many active `deadline_timer` objects.
- Fixed the kqueue backend so that it compiles on NetBSD.
- Fixed the socket `io_control()` implementation on 64-bit Mac OS X and BSD platforms.
- Fixed a Windows-specific problem where failures from `accept()` are incorrectly treated as successes.
- Deprecated the separate compilation header `asio/impl/src.cpp` in favour of `asio/impl/src.hpp`.

Asio 1.4.6

- Reduced compile times. (Note that some programs may need to add additional `#includes`, e.g. if the program uses `boost::array` but does not explicitly include `<boost/array.hpp>`.)
- Reduced the size of generated code.
- Refactored `deadline_timer` implementation to improve performance.
- Improved multiprocessor scalability on Windows by using a dedicated hidden thread to wait for timers.
- Improved performance of `asio::streambuf` with `async_read()` and `async_read_until()`. These read operations now use the existing capacity of the `streambuf` when reading, rather than limiting the read to 512 bytes.
- Added optional separate compilation. To enable, include `asio/impl/src.cpp` in one source file in a program, then build the program with `(BOOST_) ASIO_SEPARATE_COMPILATION` defined in the project/compiler settings. Alternatively, `(BOOST_) ASIO_DYN_LINK` may be defined to build a separately-compiled Asio as part of a shared library.
- Added new macro `(BOOST_) ASIO_DISABLE_FENCED_BLOCK` to permit the disabling of memory fences around completion handlers, even if thread support is enabled.
- Reworked timeout examples to better illustrate typical use cases.

- Ensured that handler arguments are passed as const types.
- Fixed incorrect parameter order in `null_buffers` variant of `async_send_to`.
- Ensured unsigned char is used with `isdigit` in `getaddrinfo` emulation.
- Fixed handling of very small but non-zero timeouts.
- Fixed crash that occurred when an empty buffer sequence was passed to a composed read or write operation.
- Added missing operator+ overload in `buffers_iterator`.
- Implemented cancellation of `null_buffers` operations on Windows.

Asio 1.4.5

- Improved performance.
- Reduced compile times.
- Reduced the size of generated code.
- Extended the guarantee that background threads don't call user code to all asynchronous operations.
- Changed to use edge-triggered epoll on Linux.
- Changed to use `timerfd` for dispatching timers on Linux, when available.
- Changed to use one-shot notifications with `kqueue` on Mac OS X and BSD platforms.
- Added a bitmask type `ip::resolver_query_base::flags` as per the TR2 proposal. This type prevents implicit conversion from `int` to `flags`, allowing the compiler to catch cases where users incorrectly pass a numeric port number as the service name.
- Added `#define NOMINMAX` for all Windows compilers. Users can define `(BOOST_)ASIO_NO_NOMINMAX` to suppress this definition.
- Fixed a bug where 0-byte asynchronous reads were incorrectly passing an `error::eof` result to the completion handler.
- Changed the `io_control()` member functions to always call `ioctl` on the underlying descriptor when modifying blocking mode.
- Changed the resolver implementation so that it no longer requires the typedefs `InternetProtocol::resolver_query` and `InternetProtocol::resolver_iterator`, as neither typedef is part of the documented `InternetProtocol` requirements. The corresponding typedefs in the `ip::tcp`, `ip::udp` and `ip::icmp` classes have been deprecated.
- Fixed out-of-band handling for reactors not based on `select()`.
- Added new `(BOOST_)ASIO_DISABLE_THREADS` macro that allows Asio's threading support to be independently disabled.
- Minor documentation improvements.

Asio 1.4.4

- Added a new HTTP Server 4 example illustrating the use of stackless coroutines with Asio.
- Changed handler allocation and invocation to use `boost::addressof` to get the address of handler objects, rather than applying `operator&` directly.
- Restricted MSVC buffer debugging workaround to 2008, as it causes a crash with 2010 beta 2.
- Fixed a problem with the lifetime of handler memory, where Windows needs the `OVERLAPPED` structure to be valid until both the initiating function call has returned and the completion packet has been delivered.

- Don't block signals while performing system calls, but instead restart the calls if they are interrupted.
- Documented the guarantee made by strand objects with respect to order of handler invocation.
- Changed strands to use a pool of implementations, to make copying of strands cheaper.
- Ensured that kqueue support is enabled for BSD platforms.
- Added a `boost_` prefix to the `extern "C"` thread entry point function.
- In `getaddrinfo` emulation, only check the socket type (`SOCK_STREAM` or `SOCK_DGRAM`) if a service name has been specified. This should allow the emulation to work with raw sockets.
- Added a workaround for some broken Windows firewalls that make a socket appear bound to 0.0.0.0 when it is in fact bound to 127.0.0.1.
- Applied a fix for reported excessive CPU usage under Solaris.
- Added some support for platforms that use older compilers such as g++ 2.95.

Asio 1.4.3

- Added a new ping example to illustrate the use of ICMP sockets.
- Changed the `buffered*_stream<>` templates to treat 0-byte reads and writes as no-ops, to comply with the documented type requirements for `SyncReadStream`, `AsyncReadStream`, `SyncWriteStream` and `AsyncWriteStream`.
- Changed some instances of the `throw` keyword to `boost::throw_exception()` to allow Asio to be used when exception support is disabled. Note that the SSL wrappers still require exception support.
- Made Asio compatible with the OpenSSL 1.0 beta.
- Eliminated a redundant system call in the Solaris /dev/poll backend.
- Fixed a bug in resizing of the bucket array in the internal hash maps.
- Ensured correct propagation of the error code when a synchronous accept fails.
- Ensured correct propagation of the error code when a synchronous read or write on a Windows HANDLE fails.
- Fixed failures reported when `_GLIBCXX_DEBUG` is defined.
- Fixed custom memory allocation support for timers.
- Tidied up various warnings reported by g++.
- Various documentation improvements, including more obvious hyperlinks to function overloads, header file information, examples for the handler type requirements, and adding enum values to the index.

Asio 1.4.2

- Implement automatic resizing of the bucket array in the internal hash maps. This is to improve performance for very large numbers of asynchronous operations and also to reduce memory usage for very small numbers. A new macro (`BOOST_`)`ASI_O_HASH_MAP_BUCKETS` may be used to tweak the sizes used for the bucket arrays. (N.B. this feature introduced a bug which was fixed in Asio 1.4.3 / Boost 1.40.)
- Add performance optimisation for the Windows IOCP backend for when no timers are used.
- Prevent locale settings from affecting formatting of TCP and UDP endpoints.
- Fix a memory leak that occurred when an asynchronous SSL operation's completion handler threw an exception.
- Fix the implementation of `io_control()` so that it adheres to the documented type requirements for `IoControlCommand`.

- Fix incompatibility between Asio and ncurses.h.
- On Windows, specifically handle the case when an overlapped `ReadFile` call fails with `ERROR_MORE_DATA`. This enables a hack where a `windows::stream_handle` can be used with a message-oriented named pipe.
- Fix system call wrappers to always clear the error on success, as POSIX allows successful system calls to modify `errno`.
- Don't include `termios.h` if `(BOOST_) ASIO_DISABLE_SERIAL_PORT` is defined.
- Cleaned up some more MSVC level 4 warnings.
- Various documentation fixes.

Asio 1.4.1

- Improved compatibility with some Windows firewall software.
- Ensured arguments to `windows::overlapped_ptr::complete()` are correctly passed to the completion handler.
- Fixed a link problem and multicast failure on QNX.
- Fixed a compile error in SSL support on MinGW / g++ 3.4.5.
- Drop back to using a pipe for notification if `eventfd` is not available at runtime on Linux.
- Various minor bug and documentation fixes.

Asio 1.4.0

- Enhanced `CompletionCondition` concept with the signature `size_t CompletionCondition(error_code ec, size_t total)`, where the return value indicates the maximum number of bytes to be transferred on the next read or write operation. (The old `CompletionCondition` signature is still supported for backwards compatibility).
- New `windows::overlapped_ptr` class to allow arbitrary overlapped I/O functions (such as `TransmitFile`) to be used with Asio.
- On recent versions of Linux, an `eventfd` descriptor is now used (rather than a pipe) to interrupt a blocked select/epoll reactor.
- Added const overloads of `lowest_layer()`.
- Synchronous read, write, accept and connect operations are now thread safe (meaning that it is now permitted to perform concurrent synchronous operations on an individual socket, if supported by the OS).
- Reactor-based `io_service` implementations now use lazy initialisation to reduce the memory usage of an `io_service` object used only as a message queue.

Asio 1.2.0

- Added support for serial ports.
- Added support for UNIX domain sockets.
- Added support for raw sockets and ICMP.
- Added wrappers for POSIX stream-oriented file descriptors (excluding regular files).
- Added wrappers for Windows stream-oriented `HANDLEs` such as named pipes (requires `HANDLEs` that work with I/O completion ports).
- Added wrappers for Windows random-access `HANDLEs` such as files (requires `HANDLEs` that work with I/O completion ports).
- Added support for reactor-style operations (i.e. they report readiness but perform no I/O) using a new `null_buffers` type.

- Added an iterator type for bytewise traversal of buffer sequences.
- Added new `read_until()` and `async_read_until()` overloads that take a user-defined function object for locating message boundaries.
- Added an experimental two-lock queue (enabled by defining `(BOOST_)ASIO_ENABLE_TWO_LOCK_QUEUE`) that may provide better `io_service` scalability across many processors.
- Various fixes, performance improvements, and more complete coverage of the custom memory allocation support.

Asio 1.0.0

First stable release of Asio.

Index

-
~basic_descriptor
 posix::basic_descriptor, 1086
~basic_handle
 windows::basic_handle, 1305
~basic_io_object
 basic_io_object, 376
~basic_socket
 basic_socket, 574
~basic_socket_streambuf
 basic_socket_streambuf, 664
~context
 ssl::context, 1233
~context_base
 ssl::context_base, 1237
~descriptor_base
 posix::descriptor_base, 1109
~error_category
 error_category, 862
~io_service
 io_service, 921
~overlapped_ptr
 windows::overlapped_ptr, 1357
~resolver_query_base
 ip::resolver_query_base, 1002
~serial_port_base
 serial_port_base, 1168
~service
 io_service::service, 924
~socket_base
 socket_base, 1202
~strand
 io_service::strand, 928
~stream
 ssl::stream, 1255
~stream_base
 ssl::stream_base, 1257
~system_error
 system_error, 1277
~thread
 thread, 1281
~work
 io_service::work, 929

A
accept
 basic_socket_acceptor, 578
 socket_acceptor_service, 1187
acceptor
 ip::tcp, 1010
 local::stream_protocol, 1050
access_denied
 error::basic_errors, 856

add
 basic_signal_set, 518
 signal_set_service, 1183
 time_traits< boost::posix_time::ptime >, 1283
add_certificate_authority
 ssl::context, 1207
add_service, 252
 io_service, 912
add_verify_path
 ssl::context, 1208
address
 ip::address, 931
 ip::basic_endpoint, 956
address_configured
 ip::basic_resolver_query, 983
 ip::resolver_query_base, 1001
address_family_not_supported
 error::basic_errors, 856
address_in_use
 error::basic_errors, 856
address_v4
 ip::address_v4, 939
address_v6
 ip::address_v6, 947
all_matching
 ip::basic_resolver_query, 983
 ip::resolver_query_base, 1001
allocator_type
 use_future_t, 1287
already_connected
 error::basic_errors, 856
already_open
 error::misc_errors, 860
already_started
 error::basic_errors, 856
any
 ip::address_v4, 940
 ip::address_v6, 948
asio_handler_allocate, 253
asio_handler_deallocate, 254
asio_handler_invoke, 254
asio_handler_is_continuation, 255
asn1
 ssl::context, 1211
 ssl::context_base, 1235
assign
 basic_datagram_socket, 297
 basic_raw_socket, 380
 basic_seq_packet_socket, 444
 basic_serial_port, 498
 basic_socket, 531
 basic_socket_acceptor, 581
 basic_socket_streambuf, 621

basic_stream_socket, 668
datagram_socket_service, 840
posix::basic_descriptor, 1071
posix::basic_stream_descriptor, 1088
posix::stream_descriptor_service, 1113
raw_socket_service, 1120
seq_packet_socket_service, 1158
serial_port_service, 1175
socket_acceptor_service, 1187
stream_socket_service, 1266
windows::basic_handle, 1296
windows::basic_object_handle, 1306
windows::basic_random_access_handle, 1319
windows::basic_stream_handle, 1335
windows::object_handle_service, 1352
windows::random_access_handle_service, 1361
windows::stream_handle_service, 1367

async_accept
 basic_socket_acceptor, 581
 socket_acceptor_service, 1187

async_connect, 256
 basic_datagram_socket, 298
 basic_raw_socket, 380
 basic_seq_packet_socket, 445
 basic_socket, 531
 basic_socket_streampbuf, 622
 basic_stream_socket, 669
 datagram_socket_service, 840
 raw_socket_service, 1120
 seq_packet_socket_service, 1158
 stream_socket_service, 1266

async_fill
 buffered_read_stream, 788
 buffered_stream, 795

async_flush
 buffered_stream, 796
 buffered_write_stream, 804

async_handshake
 ssl::stream, 1240

async_read, 263

async_read_at, 269

async_read_some
 basic_serial_port, 498
 basic_stream_socket, 670
 buffered_read_stream, 788
 buffered_stream, 796
 buffered_write_stream, 804
 posix::basic_stream_descriptor, 1089
 posix::stream_descriptor_service, 1113
 serial_port_service, 1175
 ssl::stream, 1242
 windows::basic_stream_handle, 1336
 windows::stream_handle_service, 1367

async_read_some_at
 windows::basic_random_access_handle, 1319
 windows::random_access_handle_service, 1361

async_read_until, 274
async_receive
 basic_datagram_socket, 299
 basic_raw_socket, 381
 basic_seq_packet_socket, 445
 basic_stream_socket, 670
 datagram_socket_service, 840
 raw_socket_service, 1120
 seq_packet_socket_service, 1158
 stream_socket_service, 1266

async_receive_from
 basic_datagram_socket, 300
 basic_raw_socket, 383
 datagram_socket_service, 841
 raw_socket_service, 1121

async_resolve
 ip::basic_resolver, 963
 ip::resolver_service, 1003

async_result
 async_result, 282

async_send
 basic_datagram_socket, 302
 basic_raw_socket, 385
 basic_seq_packet_socket, 447
 basic_stream_socket, 672
 datagram_socket_service, 841
 raw_socket_service, 1121
 seq_packet_socket_service, 1159
 stream_socket_service, 1267

async_send_to
 basic_datagram_socket, 304
 basic_raw_socket, 387
 datagram_socket_service, 841
 raw_socket_service, 1121

async_shutdown
 ssl::stream, 1242

async_wait
 basic_deadline_timer, 361
 basic_signal_set, 519
 basic_waitable_timer, 734
 deadline_timer_service, 852
 signal_set_service, 1183
 waitable_timer_service, 1290
 windows::basic_object_handle, 1307
 windows::object_handle_service, 1352

async_write, 283
async_write_at, 288
async_write_some
 basic_serial_port, 499
 basic_stream_socket, 674
 buffered_read_stream, 788
 buffered_stream, 796
 buffered_write_stream, 804
 posix::basic_stream_descriptor, 1090
 posix::stream_descriptor_service, 1113
 serial_port_service, 1176

ssl::stream, 1243
 windows::basic_stream_handle, 1337
 windows::stream_handle_service, 1368
 async_write_some_at
 windows::basic_random_access_handle, 1320
 windows::random_access_handle_service, 1361
 at_mark
 basic_datagram_socket, 306
 basic_raw_socket, 388
 basic_seq_packet_socket, 448
 basic_socket, 532
 basic_socket_streampbuf, 623
 basic_stream_socket, 675
 datagram_socket_service, 841
 raw_socket_service, 1121
 seq_packet_socket_service, 1159
 stream_socket_service, 1267
 available
 basic_datagram_socket, 307
 basic_raw_socket, 389
 basic_seq_packet_socket, 449
 basic_socket, 533
 basic_socket_streampbuf, 623
 basic_stream_socket, 676
 datagram_socket_service, 842
 raw_socket_service, 1122
 seq_packet_socket_service, 1159
 stream_socket_service, 1267

B

bad_descriptor
 error::basic_errors, 856
 basic_datagram_socket
 basic_datagram_socket, 308
 basic_deadline_timer
 basic_deadline_timer, 361
 basic_descriptor
 posix::basic_descriptor, 1072
 basic_endpoint
 generic::basic_endpoint, 867
 ip::basic_endpoint, 957
 local::basic_endpoint, 1037
 basic_handle
 windows::basic_handle, 1296
 basic_io_object
 basic_io_object, 373
 basic_object_handle
 windows::basic_object_handle, 1308
 basic_random_access_handle
 windows::basic_random_access_handle, 1321
 basic_raw_socket
 basic_raw_socket, 390
 basic_resolver
 ip::basic_resolver, 965
 basic_resolver_entry
 ip::basic_resolver_entry, 974
 basic_resolver_iterator
 ip::basic_resolver_iterator, 977
 basic_resolver_query
 ip::basic_resolver_query, 984
 basic_seq_packet_socket
 basic_seq_packet_socket, 450
 basic_serial_port
 basic_serial_port, 500
 basic_signal_set
 basic_signal_set, 520
 basic_socket
 basic_socket, 534
 basic_socket_acceptor
 basic_socket_acceptor, 583
 basic_socket_iostream
 basic_socket_iostream, 614
 basic_socket_streampbuf
 basic_socket_streampbuf, 624
 basic_stream_descriptor
 posix::basic_stream_descriptor, 1090
 basic_stream_handle
 windows::basic_stream_handle, 1337
 basic_stream_socket
 basic_stream_socket, 677
 basic_streampbuf
 basic_streampbuf, 728
 basic_waitable_timer
 basic_waitable_timer, 735
 basic_yield_context
 basic_yield_context, 746
 baud_rate
 serial_port_base::baud_rate, 1169
 begin
 buffers_iterator, 812
 const_buffers_1, 831
 mutable_buffers_1, 1063
 null_buffers, 1066
 bind
 basic_datagram_socket, 311
 basic_raw_socket, 393
 basic_seq_packet_socket, 453
 basic_socket, 537
 basic_socket_acceptor, 586
 basic_socket_streampbuf, 624
 basic_stream_socket, 680
 datagram_socket_service, 842
 raw_socket_service, 1122
 seq_packet_socket_service, 1159
 socket_acceptor_service, 1187
 stream_socket_service, 1267
 broadcast
 basic_datagram_socket, 312
 basic_raw_socket, 394
 basic_seq_packet_socket, 454
 basic_socket, 538
 basic_socket_acceptor, 588
 basic_socket_streampbuf, 626

basic_stream_socket, 681
ip::address_v4, 940
socket_base, 1194
broken_pipe
 error::basic_errors, 856
buffer, 747
buffer_cast, 763
buffer_copy, 764
buffer_size, 785
buffered_read_stream
 buffered_read_stream, 789
buffered_stream
 buffered_stream, 796
buffered_write_stream
 buffered_write_stream, 804
buffers_begin, 810
buffers_end, 810
buffers_iterator
 buffers_iterator, 812
bytes_readable
 basic_datagram_socket, 312
 basic_raw_socket, 395
 basic_seq_packet_socket, 455
 basic_socket, 538
 basic_socket_acceptor, 588
 basic_socket_streambuf, 626
 basic_stream_socket, 682
 posix::basic_descriptor, 1073
 posix::basic_stream_descriptor, 1092
 posix::descriptor_base, 1108
 socket_base, 1195
bytes_type
 ip::address_v4, 940
 ip::address_v6, 948

C

callee_type
 basic_yield_context, 746

caller_type
 basic_yield_context, 747

cancel
 basic_datagram_socket, 313
 basic_deadline_timer, 362
 basic_raw_socket, 395
 basic_seq_packet_socket, 455
 basic_serial_port, 502
 basic_signal_set, 522
 basic_socket, 539
 basic_socket_acceptor, 589
 basic_socket_streambuf, 627
 basic_stream_socket, 682
 basic_waitable_timer, 736
 datagram_socket_service, 842
 deadline_timer_service, 852
 ip::basic_resolver, 965
 ip::resolver_service, 1004
 posix::basic_descriptor, 1073

 posix::basic_stream_descriptor, 1092
 posix::stream_descriptor_service, 1113
 raw_socket_service, 1122
 seq_packet_socket_service, 1159
 serial_port_service, 1176
 signal_set_service, 1183
 socket_acceptor_service, 1187
 stream_socket_service, 1267
 waitable_timer_service, 1291
 windows::basic_handle, 1298
 windows::basic_object_handle, 1309
 windows::basic_random_access_handle, 1322
 windows::basic_stream_handle, 1339
 windows::object_handle_service, 1352
 windows::random_access_handle_service, 1361
 windows::stream_handle_service, 1368

cancel_one
 basic_deadline_timer, 364
 basic_waitable_timer, 737
 deadline_timer_service, 853
 waitable_timer_service, 1291

canonical_name
 ip::basic_resolver_query, 986
 ip::resolver_query_base, 1001

capacity
 generic::basic_endpoint, 868
 ip::basic_endpoint, 958
 local::basic_endpoint, 1038

category
 error_code, 863

character_size
 serial_port_base::character_size, 1169

clear
 basic_signal_set, 523
 signal_set_service, 1183

clear_options
 ssl::context, 1209

client
 ssl::stream, 1245
 ssl::stream_base, 1257

clock_type
 basic_waitable_timer, 738
 waitable_timer_service, 1291

close
 basic_datagram_socket, 314
 basic_raw_socket, 397
 basic_seq_packet_socket, 457
 basic_serial_port, 503
 basic_socket, 540
 basic_socket_acceptor, 589
 basic_socket_iostream, 614
 basic_socket_streambuf, 628
 basic_stream_socket, 684
 buffered_read_stream, 789
 buffered_stream, 797
 buffered_write_stream, 805

datagram_socket_service, 842
 posix::basic_descriptor, 1074
 posix::basic_stream_descriptor, 1093
 posix::stream_descriptor_service, 1114
 raw_socket_service, 1122
 seq_packet_socket_service, 1160
 serial_port_service, 1176
 socket_acceptor_service, 1188
 stream_socket_service, 1267
 windows::basic_handle, 1298
 windows::basic_object_handle, 1310
 windows::basic_random_access_handle, 1323
 windows::basic_stream_handle, 1339
 windows::object_handle_service, 1352
 windows::random_access_handle_service, 1361
 windows::stream_handle_service, 1368
code
 system_error, 1276
commit
 basic_streambuf, 728
complete
 windows::overlapped_ptr, 1355
connect, 818
 basic_datagram_socket, 315
 basic_raw_socket, 398
 basic_seq_packet_socket, 458
 basic_socket, 541
 basic_socket_iostream, 614
 basic_socket_streambuf, 629
 basic_stream_socket, 685
 datagram_socket_service, 842
 raw_socket_service, 1122
 seq_packet_socket_service, 1160
 stream_socket_service, 1268
connection_aborted
 error::basic_errors, 856
connection_refused
 error::basic_errors, 856
connection_reset
 error::basic_errors, 856
const_buffer
 const_buffer, 829
const_buffers_1
 const_buffers_1, 831
const_buffers_type
 basic_streambuf, 729
const_iterator
 const_buffers_1, 832
 mutable_buffers_1, 1063
 null_buffers, 1066
construct
 datagram_socket_service, 842
 deadline_timer_service, 853
 ip::resolver_service, 1004
 posix::stream_descriptor_service, 1114
 raw_socket_service, 1122
 seq_packet_socket_service, 1160
 serial_port_service, 1176
 signal_set_service, 1184
 socket_acceptor_service, 1188
 stream_socket_service, 1268
 waitable_timer_service, 1291
 windows::object_handle_service, 1352
 windows::random_access_handle_service, 1362
 windows::stream_handle_service, 1368
consume
 basic_streambuf, 729
context
 ssl::context, 1210
converting_move_construct
 datagram_socket_service, 843
 raw_socket_service, 1123
 seq_packet_socket_service, 1160
 socket_acceptor_service, 1188
 stream_socket_service, 1268
coroutine
 coroutine, 837
create
 ip::basic_resolver_iterator, 977

D

data
 basic_streambuf, 729
 generic::basic_endpoint, 869
 ip::basic_endpoint, 958
 local::basic_endpoint, 1038
data_type
 generic::basic_endpoint, 869
 ip::basic_endpoint, 958
 local::basic_endpoint, 1038
datagram_protocol
 generic::datagram_protocol, 873
datagram_socket_service
 datagram_socket_service, 843
deadline_timer, 848
deadline_timer_service
 deadline_timer_service, 853
debug
 basic_datagram_socket, 317
 basic_raw_socket, 399
 basic_seq_packet_socket, 459
 basic_socket, 543
 basic_socket_acceptor, 590
 basic_socket_streambuf, 631
 basic_stream_socket, 686
 socket_base, 1195
default_buffer_size
 buffered_read_stream, 790
 buffered_write_stream, 805
default_workarounds
 ssl::context, 1211
 ssl::context_base, 1234
destroy

datagram_socket_service, 843
 deadline_timer_service, 853
 ip::resolver_service, 1004
 posix::stream_descriptor_service, 1114
 raw_socket_service, 1123
 seq_packet_socket_service, 1160
 serial_port_service, 1176
 signal_set_service, 1184
 socket_acceptor_service, 1188
 stream_socket_service, 1268
 waitable_timer_service, 1291
 windows::object_handle_service, 1353
 windows::random_access_handle_service, 1362
 windows::stream_handle_service, 1368
 difference_type
 buffers_iterator, 812
 ip::basic_resolver_iterator, 978
 dispatch
 io_service, 912
 io_service::strand, 926
 do_not_route
 basic_datagram_socket, 317
 basic_raw_socket, 400
 basic_seq_packet_socket, 460
 basic_socket, 543
 basic_socket_acceptor, 591
 basic_socket_streambuf, 631
 basic_stream_socket, 687
 socket_base, 1196
 duration
 basic_waitable_timer, 739
 waitable_timer_service, 1291
 duration_type
 basic_deadline_timer, 365
 basic_socket_iostream, 615
 basic_socket_streambuf, 632
 deadline_timer_service, 853
 time_traits< boost::posix_time::ptime >, 1283

E

enable_connection_aborted
 basic_datagram_socket, 318
 basic_raw_socket, 401
 basic_seq_packet_socket, 460
 basic_socket, 544
 basic_socket_acceptor, 591
 basic_socket_streambuf, 632
 basic_stream_socket, 687
 socket_base, 1196

end

 buffers_iterator, 812
 const_buffers_1, 832
 mutable_buffers_1, 1063
 null_buffers, 1067

endpoint

 generic::datagram_protocol, 873
 generic::raw_protocol, 881

 generic::seq_packet_protocol, 888
 generic::stream_protocol, 896
 ip::basic_resolver_entry, 974
 ip::icmp, 989
 ip::tcp, 1013
 ip::udp, 1024
 local::datagram_protocol, 1043
 local::stream_protocol, 1053

endpoint_type

 basic_datagram_socket, 319
 basic_raw_socket, 401
 basic_seq_packet_socket, 461
 basic_socket, 544
 basic_socket_acceptor, 592
 basic_socket_iostream, 615
 basic_socket_streambuf, 632
 basic_stream_socket, 688
 datagram_socket_service, 843
 ip::basic_resolver, 965
 ip::basic_resolver_entry, 974
 ip::resolver_service, 1004
 raw_socket_service, 1123
 seq_packet_socket_service, 1160
 socket_acceptor_service, 1188
 stream_socket_service, 1268

eof

 error::misc_errors, 860

error

 basic_socket_iostream, 615
 basic_socket_streambuf, 633

 error::addrinfo_category, 856
 error::addrinfo_errors, 856
 error::basic_errors, 856
 error::get_addrinfo_category, 858
 error::get_misc_category, 858
 error::get_netdb_category, 858
 error::get_ssl_category, 858
 error::get_system_category, 858
 error::make_error_code, 859
 error::misc_category, 860
 error::misc_errors, 860
 error::netdb_category, 860
 error::netdb_errors, 860
 error::ssl_category, 861
 error::ssl_errors, 861
 error::system_category, 861

error_code

 error_code, 864

even

 serial_port_base::parity, 1172

expires_at

 basic_deadline_timer, 365
 basic_socket_iostream, 616
 basic_socket_streambuf, 633
 basic_waitable_timer, 739
 deadline_timer_service, 854

waitable_timer_service, 1292
expires_from_now
 basic_deadline_timer, 367
 basic_socket_iostream, 616
 basic_socket_streambuf, 634
 basic_waitable_timer, 741
 deadline_timer_service, 854
 waitable_timer_service, 1292

F
family
 generic::datagram_protocol, 875
 generic::raw_protocol, 882
 generic::seq_packet_protocol, 890
 generic::stream_protocol, 898
 ip::icmp, 991
 ip::tcp, 1015
 ip::udp, 1026
 local::datagram_protocol, 1045
 local::stream_protocol, 1055

fault
 error::basic_errors, 856

fd_set_failure
 error::misc_errors, 860

file_format
 ssl::context, 1211
 ssl::context_base, 1235

fill
 buffered_read_stream, 790
 buffered_stream, 797

flags
 ip::basic_resolver_query, 986
 ip::resolver_query_base, 1001

flow_control
 serial_port_base::flow_control, 1170

flush
 buffered_stream, 798
 buffered_write_stream, 806

for_reading
 ssl::context, 1215
 ssl::context_base, 1237

for_writing
 ssl::context, 1215
 ssl::context_base, 1237

fork_child
 io_service, 913

fork_event
 io_service, 913

fork_parent
 io_service, 913

fork_prepare
 io_service, 913

fork_service
 io_service::service, 924

from_string
 ip::address, 932
 ip::address_v4, 941

ip::address_v6, 948

G
get
 async_result, 283
 windows::overlapped_ptr, 1355

get_allocator
 use_future_t, 1287

get_implementation
 basic_datagram_socket, 319
 basic_deadline_timer, 368
 basic_io_object, 374
 basic_raw_socket, 401
 basic_seq_packet_socket, 461
 basic_serial_port, 503
 basic_signal_set, 524
 basic_socket, 545
 basic_socket_acceptor, 592
 basic_socket_streambuf, 634
 basic_stream_socket, 688
 basic_waitable_timer, 742
 ip::basic_resolver, 965
 posix::basic_descriptor, 1075
 posix::basic_stream_descriptor, 1094
 windows::basic_handle, 1299
 windows::basic_object_handle, 1310
 windows::basic_random_access_handle, 1324
 windows::basic_stream_handle, 1340

get_io_service
 basic_datagram_socket, 319
 basic_deadline_timer, 369
 basic_io_object, 374
 basic_raw_socket, 402
 basic_seq_packet_socket, 462
 basic_serial_port, 504
 basic_signal_set, 525
 basic_socket, 545
 basic_socket_acceptor, 593
 basic_socket_streambuf, 635
 basic_stream_socket, 689
 basic_waitable_timer, 743
 buffered_read_stream, 790
 buffered_stream, 798
 buffered_write_stream, 806
 datagram_socket_service, 843
 deadline_timer_service, 855
 io_service::service, 923
 io_service::strand, 926
 io_service::work, 929
 ip::basic_resolver, 966
 ip::resolver_service, 1005
 posix::basic_descriptor, 1075
 posix::basic_stream_descriptor, 1094
 posix::stream_descriptor_service, 1114
 raw_socket_service, 1123
 seq_packet_socket_service, 1161
 serial_port_service, 1176

signal_set_service, 1184
socket_acceptor_service, 1188
ssl::stream, 1243
stream_socket_service, 1268
waitable_timer_service, 1293
windows::basic_handle, 1299
windows::basic_object_handle, 1311
windows::basic_random_access_handle, 1324
windows::basic_stream_handle, 1341
windows::object_handle_service, 1353
windows::random_access_handle_service, 1362
windows::stream_handle_service, 1368
get_option
 basic_datagram_socket, 320
 basic_raw_socket, 402
 basic_seq_packet_socket, 462
 basic_serial_port, 504
 basic_socket, 545
 basic_socket_acceptor, 593
 basic_socket_streambuf, 635
 basic_stream_socket, 689
 datagram_socket_service, 843
 raw_socket_service, 1123
 seq_packet_socket_service, 1161
 serial_port_service, 1177
 socket_acceptor_service, 1189
 stream_socket_service, 1269
get_service
 basic_datagram_socket, 321
 basic_deadline_timer, 369
 basic_io_object, 374
 basic_raw_socket, 404
 basic_seq_packet_socket, 463
 basic_serial_port, 505
 basic_signal_set, 525
 basic_socket, 547
 basic_socket_acceptor, 594
 basic_socket_streambuf, 636
 basic_stream_socket, 690
 basic_waitable_timer, 743
 ip::basic_resolver, 966
 posix::basic_descriptor, 1076
 posix::basic_stream_descriptor, 1094
 windows::basic_handle, 1300
 windows::basic_object_handle, 1311
 windows::basic_random_access_handle, 1325
 windows::basic_stream_handle, 1341

H
handshake
 ssl::stream, 1243
handshake_type
 ssl::stream, 1245
 ssl::stream_base, 1257
hardware
 serial_port_base::flow_control, 1171
has_service, 905

io_service, 913
high_resolution_timer, 905
hints
 ip::basic_resolver_query, 987
host_name
 ip::basic_resolver_entry, 975
 ip::basic_resolver_query, 987
host_not_found
 error::netdb_errors, 860
host_not_found_try_again
 error::netdb_errors, 860
host_unreachable
 error::basic_errors, 856

I
id
 datagram_socket_service, 844
 deadline_timer_service, 855
 io_service::id, 922
 ip::resolver_service, 1005
 posix::stream_descriptor_service, 1114
 raw_socket_service, 1123
 seq_packet_socket_service, 1161
 serial_port_service, 1177
 signal_set_service, 1184
 socket_acceptor_service, 1189
 stream_socket_service, 1269
 waitable_timer_service, 1293
 windows::object_handle_service, 1353
 windows::random_access_handle_service, 1362
 windows::stream_handle_service, 1369
impl
 ssl::context, 1212
 ssl::stream, 1246
impl_type
 ssl::context, 1212
 ssl::stream, 1246
implementation
 basic_datagram_socket, 321
 basic_deadline_timer, 370
 basic_io_object, 375
 basic_raw_socket, 404
 basic_seq_packet_socket, 464
 basic_serial_port, 506
 basic_signal_set, 525
 basic_socket, 547
 basic_socket_acceptor, 595
 basic_socket_streambuf, 637
 basic_stream_socket, 691
 basic_waitable_timer, 743
 ip::basic_resolver, 967
 posix::basic_descriptor, 1076
 posix::basic_stream_descriptor, 1095
 windows::basic_handle, 1300
 windows::basic_object_handle, 1312
 windows::basic_random_access_handle, 1325
 windows::basic_stream_handle, 1341

implementation_type
 basic_datagram_socket, 322
 basic_deadline_timer, 370
 basic_io_object, 375
 basic_raw_socket, 404
 basic_seq_packet_socket, 464
 basic_serial_port, 506
 basic_signal_set, 525
 basic_socket, 547
 basic_socket_acceptor, 595
 basic_socket_streambuf, 637
 basic_stream_socket, 691
 basic_waitable_timer, 743
 datagram_socket_service, 844
 deadline_timer_service, 855
 ip::basic_resolver, 967
 ip::resolver_service, 1005
 posix::basic_descriptor, 1076
 posix::basic_stream_descriptor, 1095
 posix::stream_descriptor_service, 1114
 raw_socket_service, 1124
 seq_packet_socket_service, 1161
 serial_port_service, 1177
 signal_set_service, 1184
 socket_acceptor_service, 1189
 stream_socket_service, 1269
 waitable_timer_service, 1293
 windows::basic_handle, 1300
 windows::basic_object_handle, 1312
 windows::basic_random_access_handle, 1325
 windows::basic_stream_handle, 1341
 windows::object_handle_service, 1353
 windows::random_access_handle_service, 1362
 windows::stream_handle_service, 1369

in_avail
 buffered_read_stream, 790
 buffered_stream, 799
 buffered_write_stream, 806

in_progress
 error::basic_errors, 856

interrupted
 error::basic_errors, 856

invalid_argument
 error::basic_errors, 856

invalid_service_owner
 invalid_service_owner, 908

io_control
 basic_datagram_socket, 322
 basic_raw_socket, 404
 basic_seq_packet_socket, 464
 basic_socket, 548
 basic_socket_acceptor, 595
 basic_socket_streambuf, 637
 basic_stream_socket, 691
 datagram_socket_service, 844
 posix::basic_descriptor, 1077

posix::basic_stream_descriptor, 1095
 posix::stream_descriptor_service, 1115
 raw_socket_service, 1124
 seq_packet_socket_service, 1161
 socket_acceptor_service, 1189
 stream_socket_service, 1269

io_handler
 basic_socket_streambuf, 639

io_service
 io_service, 914

iostream
 generic::stream_protocol, 898
 ip::tcp, 1015
 local::stream_protocol, 1055

ip::host_name, 988

ip::multicast::enable_loopback, 997

ip::multicast::hops, 998

ip::multicast::join_group, 999

ip::multicast::leave_group, 999

ip::multicast::outbound_interface, 999

ip::unicast::hops, 1032

ip::v6_only, 1033

is_child
 coroutine, 837

is_class_a
 ip::address_v4, 941

is_class_b
 ip::address_v4, 942

is_class_c
 ip::address_v4, 942

is_complete
 coroutine, 838

is_link_local
 ip::address_v6, 949

is_loopback
 ip::address, 933
 ip::address_v4, 942
 ip::address_v6, 949

is_multicast
 ip::address, 933
 ip::address_v4, 942
 ip::address_v6, 950

is_multicast_global
 ip::address_v6, 950

is_multicast_link_local
 ip::address_v6, 950

is_multicast_node_local
 ip::address_v6, 950

is_multicast_org_local
 ip::address_v6, 950

is_multicast_site_local
 ip::address_v6, 950

is_open
 basic_datagram_socket, 323
 basic_raw_socket, 406
 basic_seq_packet_socket, 466

basic_serial_port, 506
basic_socket, 549
basic_socket_acceptor, 597
basic_socket_streambuf, 639
basic_stream_socket, 693
datagram_socket_service, 844
posix::basic_descriptor, 1078
posix::basic_stream_descriptor, 1097
posix::stream_descriptor_service, 1115
raw_socket_service, 1124
seq_packet_socket_service, 1161
serial_port_service, 1177
socket_acceptor_service, 1189
stream_socket_service, 1269
windows::basic_handle, 1300
windows::basic_object_handle, 1312
windows::basic_random_access_handle, 1325
windows::basic_stream_handle, 1342
windows::object_handle_service, 1353
windows::random_access_handle_service, 1362
windows::stream_handle_service, 1369
is_parent
 coroutine, 838
is_site_local
 ip::address_v6, 950
is_unspecified
 ip::address, 933
 ip::address_v4, 942
 ip::address_v6, 950
is_v4
 ip::address, 933
is_v4_compatible
 ip::address_v6, 951
is_v4_mapped
 ip::address_v6, 951
is_v6
 ip::address, 933
iterator
 ip::basic_resolver, 967
iterator_category
 buffers_iterator, 812
 ip::basic_resolver_iterator, 978
iterator_type
 ip::resolver_service, 1005

J
join
 thread, 1281

K
keep_alive
 basic_datagram_socket, 323
 basic_raw_socket, 406
 basic_seq_packet_socket, 466
 basic_socket, 549
 basic_socket_acceptor, 597
 basic_socket_streambuf, 639
basic_stream_socket, 693
socket_base, 1197

L
less_than
 time_traits< boost::posix_time::ptime >, 1283
linger
 basic_datagram_socket, 324
 basic_raw_socket, 407
 basic_seq_packet_socket, 467
 basic_socket, 550
 basic_socket_acceptor, 597
 basic_socket_streambuf, 640
 basic_stream_socket, 694
 socket_base, 1197
listen
 basic_socket_acceptor, 598
 socket_acceptor_service, 1189
load
 serial_port_base::baud_rate, 1169
 serial_port_base::character_size, 1170
 serial_port_base::flow_control, 1171
 serial_port_base::parity, 1172
 serial_port_base::stop_bits, 1173
load_verify_file
 ssl::context, 1212
local::connect_pair, 1042
local_endpoint
 basic_datagram_socket, 325
 basic_raw_socket, 407
 basic_seq_packet_socket, 467
 basic_socket, 551
 basic_socket_acceptor, 599
 basic_socket_streambuf, 640
 basic_stream_socket, 694
 datagram_socket_service, 844
 raw_socket_service, 1124
 seq_packet_socket_service, 1162
 socket_acceptor_service, 1190
 stream_socket_service, 1269
loopback
 ip::address_v4, 942
 ip::address_v6, 951
lowest_layer
 basic_datagram_socket, 326
 basic_raw_socket, 408
 basic_seq_packet_socket, 468
 basic_serial_port, 506
 basic_socket, 552
 basic_socket_streambuf, 641
 basic_stream_socket, 695
 buffered_read_stream, 791
 buffered_stream, 799
 buffered_write_stream, 807
 posix::basic_descriptor, 1078
 posix::basic_stream_descriptor, 1097
 ssl::stream, 1246

windows::basic_handle, 1301
 windows::basic_object_handle, 1312
 windows::basic_random_access_handle, 1326
 windows::basic_stream_handle, 1342

lowest_layer_type
 basic_datagram_socket, 327
 basic_raw_socket, 409
 basic_seq_packet_socket, 469
 basic_serial_port, 507
 basic_socket, 552
 basic_socket_streampbuf, 642
 basic_stream_socket, 696
 buffered_read_stream, 791
 buffered_stream, 799
 buffered_write_stream, 807
 posix::basic_descriptor, 1079
 posix::basic_stream_descriptor, 1097
 ssl::stream, 1247
 windows::basic_handle, 1301
 windows::basic_object_handle, 1313
 windows::basic_random_access_handle, 1326
 windows::basic_stream_handle, 1343

M

max_connections
 basic_datagram_socket, 330
 basic_raw_socket, 413
 basic_seq_packet_socket, 473
 basic_socket, 556
 basic_socket_acceptor, 600
 basic_socket_streampbuf, 646
 basic_stream_socket, 700
 socket_base, 1198

max_size
 basic_streampbuf, 729

message
 error_category, 862
 error_code, 864

message_do_not_route
 basic_datagram_socket, 330
 basic_raw_socket, 413
 basic_seq_packet_socket, 473
 basic_socket, 556
 basic_socket_acceptor, 600
 basic_socket_streampbuf, 646
 basic_stream_socket, 700
 socket_base, 1198

message_end_of_record
 basic_datagram_socket, 330
 basic_raw_socket, 413
 basic_seq_packet_socket, 473
 basic_socket, 556
 basic_socket_acceptor, 601
 basic_socket_streampbuf, 646
 basic_stream_socket, 700
 socket_base, 1198

message_flags
 basic_datagram_socket, 330
 basic_raw_socket, 413
 basic_seq_packet_socket, 473
 basic_socket, 556
 basic_socket_acceptor, 601
 basic_socket_streampbuf, 646
 basic_stream_socket, 700
 socket_base, 1198

message_out_of_band
 basic_datagram_socket, 331
 basic_raw_socket, 413
 basic_seq_packet_socket, 473
 basic_socket, 557
 basic_socket_acceptor, 601
 basic_socket_streampbuf, 646
 basic_stream_socket, 700
 socket_base, 1198

message_peek
 basic_datagram_socket, 331
 basic_raw_socket, 414
 basic_seq_packet_socket, 474
 basic_socket, 557
 basic_socket_acceptor, 601
 basic_socket_streampbuf, 647
 basic_stream_socket, 701
 socket_base, 1199

message_size
 error::basic_errors, 856

method
 ssl::context, 1213
 ssl::context_base, 1235

move_assign
 datagram_socket_service, 844
 posix::stream_descriptor_service, 1115
 raw_socket_service, 1124
 seq_packet_socket_service, 1162
 serial_port_service, 1177
 socket_acceptor_service, 1190
 stream_socket_service, 1270
 windows::object_handle_service, 1353
 windows::random_access_handle_service, 1362
 windows::stream_handle_service, 1369

move_construct
 datagram_socket_service, 845
 posix::stream_descriptor_service, 1115
 raw_socket_service, 1124
 seq_packet_socket_service, 1162
 serial_port_service, 1177
 socket_acceptor_service, 1190
 stream_socket_service, 1270
 windows::object_handle_service, 1354
 windows::random_access_handle_service, 1363
 windows::stream_handle_service, 1369

mutable_buffer
 mutable_buffer, 1061

mutable_buffers_1

mutable_buffers_1, 1064
mutable_buffers_type
 basic_streampbuf, 730

N

name
 error_category, 862
name_too_long
 error::basic_errors, 856
native
 basic_datagram_socket, 331
 basic_raw_socket, 414
 basic_seq_packet_socket, 474
 basic_serial_port, 509
 basic_socket, 557
 basic_socket_acceptor, 601
 basic_socket_streampbuf, 647
 basic_stream_socket, 701
 datagram_socket_service, 845
 posix::basic_descriptor, 1081
 posix::basic_stream_descriptor, 1099
 posix::stream_descriptor_service, 1115
 raw_socket_service, 1125
 seq_packet_socket_service, 1162
 serial_port_service, 1178
 socket_acceptor_service, 1190
 stream_socket_service, 1270
 windows::basic_handle, 1303
 windows::basic_object_handle, 1315
 windows::basic_random_access_handle, 1328
 windows::basic_stream_handle, 1344
 windows::random_access_handle_service, 1363
 windows::stream_handle_service, 1369

native_handle
 basic_datagram_socket, 331
 basic_raw_socket, 414
 basic_seq_packet_socket, 474
 basic_serial_port, 509
 basic_socket, 557
 basic_socket_acceptor, 601
 basic_socket_streampbuf, 647
 basic_stream_socket, 701
 datagram_socket_service, 845
 posix::basic_descriptor, 1081
 posix::basic_stream_descriptor, 1100
 posix::stream_descriptor_service, 1115
 raw_socket_service, 1125
 seq_packet_socket_service, 1162
 socket_acceptor_service, 1190
 ssl::context, 1214
 ssl::stream, 1247
 ssl::verify_context, 1258
 stream_socket_service, 1270
 windows::basic_handle, 1303
 windows::basic_object_handle, 1315
 windows::basic_random_access_handle, 1328

windows::basic_stream_handle, 1344
windows::object_handle_service, 1354
windows::random_access_handle_service, 1363
windows::stream_handle_service, 1370

native_handle_type
 basic_datagram_socket, 331
 basic_raw_socket, 414
 basic_seq_packet_socket, 474
 basic_serial_port, 509
 basic_socket, 557
 basic_socket_acceptor, 602
 basic_socket_streampbuf, 647
 basic_stream_socket, 701
 datagram_socket_service, 845
 posix::basic_descriptor, 1081
 posix::basic_stream_descriptor, 1100
 posix::stream_descriptor_service, 1115
 raw_socket_service, 1125
 seq_packet_socket_service, 1162
 serial_port_service, 1178
 socket_acceptor_service, 1190
 ssl::context, 1214
 ssl::stream, 1247
 ssl::verify_context, 1258
 stream_socket_service, 1270
 windows::basic_handle, 1303
 windows::basic_object_handle, 1315
 windows::basic_random_access_handle, 1329
 windows::basic_stream_handle, 1345
 windows::object_handle_service, 1354
 windows::random_access_handle_service, 1363
 windows::stream_handle_service, 1370

native_non_blocking
 basic_datagram_socket, 332
 basic_raw_socket, 414
 basic_seq_packet_socket, 474
 basic_socket, 557
 basic_socket_acceptor, 602
 basic_socket_streampbuf, 647
 basic_stream_socket, 701
 datagram_socket_service, 845
 posix::basic_descriptor, 1081
 posix::basic_stream_descriptor, 1100
 posix::stream_descriptor_service, 1116
 raw_socket_service, 1125
 seq_packet_socket_service, 1163
 socket_acceptor_service, 1191
 stream_socket_service, 1270

native_type
 basic_datagram_socket, 337
 basic_raw_socket, 419
 basic_seq_packet_socket, 479
 basic_serial_port, 509
 basic_socket, 562
 basic_socket_acceptor, 603
 basic_socket_streampbuf, 652

basic_stream_socket, 706
 datagram_socket_service, 846
 posix::basic_descriptor, 1083
 posix::basic_stream_descriptor, 1101
 posix::stream_descriptor_service, 1116
 raw_socket_service, 1126
 seq_packet_socket_service, 1163
 serial_port_service, 1178
 socket_acceptor_service, 1191
 stream_socket_service, 1271
 windows::basic_handle, 1304
 windows::basic_object_handle, 1315
 windows::basic_random_access_handle, 1329
 windows::basic_stream_handle, 1345
 windows::random_access_handle_service, 1363
 windows::stream_handle_service, 1370
netmask
 ip::address_v4, 942
network_down
 error::basic_errors, 856
network_reset
 error::basic_errors, 856
network_unreachable
 error::basic_errors, 856
next_layer
 buffered_read_stream, 791
 buffered_stream, 800
 buffered_write_stream, 807
 ssl::stream, 1247
next_layer_type
 buffered_read_stream, 791
 buffered_stream, 800
 buffered_write_stream, 807
 ssl::stream, 1248
no_buffer_space
 error::basic_errors, 856
no_compression
 ssl::context, 1214
 ssl::context_base, 1236
no_data
 error::netdb_errors, 860
no_delay
 ip::tcp, 1016
no_descriptors
 error::basic_errors, 856
no_memory
 error::basic_errors, 856
no_permission
 error::basic_errors, 856
no_protocol_option
 error::basic_errors, 856
no_recovery
 error::netdb_errors, 860
no_sslv2
 ssl::context, 1214
 ssl::context_base, 1236
no_sslv3
 ssl::context, 1214
 ssl::context_base, 1236
no_such_device
 error::basic_errors, 856
no_tls1
 ssl::context, 1214
 ssl::context_base, 1236
no_tls1_1
 ssl::context, 1214
 ssl::context_base, 1236
no_tls1_2
 ssl::context, 1215
 ssl::context_base, 1236
non_blocking
 basic_datagram_socket, 337
 basic_raw_socket, 419
 basic_seq_packet_socket, 479
 basic_socket, 563
 basic_socket_acceptor, 603
 basic_socket_streambuf, 652
 basic_stream_socket, 706
 datagram_socket_service, 846
 posix::basic_descriptor, 1083
 posix::basic_stream_descriptor, 1102
 posix::stream_descriptor_service, 1117
 raw_socket_service, 1126
 seq_packet_socket_service, 1163
 socket_acceptor_service, 1191
 stream_socket_service, 1271
non_blocking_io
 basic_datagram_socket, 338
 basic_raw_socket, 421
 basic_seq_packet_socket, 481
 basic_socket, 564
 basic_socket_acceptor, 605
 basic_socket_streambuf, 654
 basic_stream_socket, 708
 posix::basic_descriptor, 1084
 posix::basic_stream_descriptor, 1103
 posix::descriptor_base, 1109
 socket_base, 1199
none
 serial_port_base::flow_control, 1171
 serial_port_base::parity, 1172
not_connected
 error::basic_errors, 856
not_found
 error::misc_errors, 860
not_socket
 error::basic_errors, 856
notify_fork
 io_service, 914
now
 time_traits< boost::posix_time::ptime >, 1284
 numeric_host

ip::basic_resolver_query, 987
 ip::resolver_query_base, 1002
numeric_service
 ip::basic_resolver_query, 987
 ip::resolver_query_base, 1002

O

object_handle_service
 windows::object_handle_service, 1354

odd
 serial_port_base::parity, 1172

one
 serial_port_base::stop_bits, 1173

onepointfive
 serial_port_base::stop_bits, 1173

open
 basic_datagram_socket, 339
 basic_raw_socket, 421
 basic_seq_packet_socket, 481
 basic_serial_port, 510
 basic_socket, 564
 basic_socket_acceptor, 605
 basic_socket_streambuf, 654
 basic_stream_socket, 708
 datagram_socket_service, 847
 raw_socket_service, 1126
 seq_packet_socket_service, 1164
 serial_port_service, 1178
 socket_acceptor_service, 1192
 stream_socket_service, 1272

operation_aborted
 error::basic_errors, 856

operation_not_supported
 error::basic_errors, 856

operator *
 buffers_iterator, 813
 ip::basic_resolver_iterator, 978

operator endpoint_type
 ip::basic_resolver_entry, 975

operator unspecified_bool_type
 error_code, 864

operator!
 error_code, 865

operator!=
 buffers_iterator, 813
 error_category, 862
 error_code, 865
 generic::basic_endpoint, 869
 generic::datagram_protocol, 875
 generic::raw_protocol, 882
 generic::seq_packet_protocol, 890
 generic::stream_protocol, 899
 ip::address, 933
 ip::address_v4, 942
 ip::address_v6, 951
 ip::basic_endpoint, 959
 ip::basic_resolver_iterator, 979

ip::icmp, 991
 ip::tcp, 1017
 ip::udp, 1026
 local::basic_endpoint, 1038

operator()
 ssl::rfc2818_verification, 1238

operator+
 buffers_iterator, 813
 const_buffer, 830
 const_buffers_1, 832
 mutable_buffer, 1062
 mutable_buffers_1, 1064

operator++
 buffers_iterator, 814
 ip::basic_resolver_iterator, 979

operator+=
 buffers_iterator, 814

operator-
 buffers_iterator, 814

operator--
 buffers_iterator, 815

operator-=
 buffers_iterator, 816

operator->
 buffers_iterator, 816
 ip::basic_resolver_iterator, 979

operator<
 buffers_iterator, 816
 generic::basic_endpoint, 869
 ip::address, 934
 ip::address_v4, 943
 ip::address_v6, 951
 ip::basic_endpoint, 959
 local::basic_endpoint, 1039

operator<<, 1067
 ip::address, 934
 ip::address_v4, 943
 ip::address_v6, 951
 ip::basic_endpoint, 959
 local::basic_endpoint, 1039

operator<=
 buffers_iterator, 816
 generic::basic_endpoint, 870
 ip::address, 934
 ip::address_v4, 943
 ip::address_v6, 952
 ip::basic_endpoint, 959
 local::basic_endpoint, 1039

operator=
 basic_datagram_socket, 340
 basic_io_object, 375
 basic_raw_socket, 423
 basic_seq_packet_socket, 483
 basic_serial_port, 510
 basic_socket, 566
 basic_socket_acceptor, 606

basic_stream_socket, 710
 generic::basic_endpoint, 870
 ip::address, 935
 ip::address_v4, 944
 ip::address_v6, 952
 ip::basic_endpoint, 960
 local::basic_endpoint, 1039
 posix::basic_descriptor, 1085
 posix::basic_stream_descriptor, 1104
 ssl::context, 1215
 system_error, 1276
 windows::basic_handle, 1304
 windows::basic_object_handle, 1315
 windows::basic_random_access_handle, 1329
 windows::basic_stream_handle, 1345

operator==
 buffers_iterator, 816
 error_category, 862
 error_code, 865
 generic::basic_endpoint, 870
 generic::datagram_protocol, 875
 generic::raw_protocol, 882
 generic::seq_packet_protocol, 890
 generic::stream_protocol, 899
 ip::address, 935
 ip::address_v4, 944
 ip::address_v6, 952
 ip::basic_endpoint, 960
 ip::basic_resolver_iterator, 979
 ip::icmp, 991
 ip::tcp, 1017
 ip::udp, 1026
 local::basic_endpoint, 1040

operator>
 buffers_iterator, 817
 generic::basic_endpoint, 870
 ip::address, 936
 ip::address_v4, 944
 ip::address_v6, 952
 ip::basic_endpoint, 960
 local::basic_endpoint, 1040

operator>=
 buffers_iterator, 817
 generic::basic_endpoint, 871
 ip::address, 936
 ip::address_v4, 944
 ip::address_v6, 953
 ip::basic_endpoint, 960
 local::basic_endpoint, 1040

operator[]
 basic_yield_context, 747
 buffers_iterator, 817
 use_future_t, 1287

options
 ssl::context, 1215
 ssl::context_base, 1236

overflow
 basic_socket_streampbuf, 656
 basic_streampbuf, 730

overlapped_ptr
 windows::overlapped_ptr, 1356

P

parity
 serial_port_base::parity, 1172

passive
 ip::basic_resolver_query, 987
 ip::resolver_query_base, 1002

password_purpose
 ssl::context, 1215
 ssl::context_base, 1237

path
 local::basic_endpoint, 1040

peek
 buffered_read_stream, 792
 buffered_stream, 800
 buffered_write_stream, 808

pem
 ssl::context, 1211
 ssl::context_base, 1235

placeholders::bytes_transferred, 1068

placeholders::error, 1068

placeholders::iterator, 1068

placeholders::signal_number, 1069

pointer
 buffers_iterator, 817
 ip::basic_resolver_iterator, 980

poll
 io_service, 915

poll_one
 io_service, 916

port
 ip::basic_endpoint, 961

posix::stream_descriptor, 1109

post
 io_service, 917
 io_service::strand, 926

prepare
 basic_streampbuf, 730

protocol
 generic::basic_endpoint, 871
 generic::datagram_protocol, 875
 generic::raw_protocol, 883
 generic::seq_packet_protocol, 890
 generic::stream_protocol, 899
 ip::basic_endpoint, 961
 ip::icmp, 992
 ip::tcp, 1017
 ip::udp, 1027
 local::basic_endpoint, 1041
 local::datagram_protocol, 1045
 local::stream_protocol, 1056

protocol_type

basic_datagram_socket, 341
 basic_raw_socket, 424
 basic_seq_packet_socket, 484
 basic_socket, 567
 basic_socket_acceptor, 607
 basic_socket_streambuf, 656
 basic_stream_socket, 711
 datagram_socket_service, 847
 generic::basic_endpoint, 871
 ip::basic_endpoint, 961
 ip::basic_resolver, 968
 ip::basic_resolver_entry, 975
 ip::basic_resolver_query, 987
 ip::resolver_service, 1006
 local::basic_endpoint, 1041
 raw_socket_service, 1126
 seq_packet_socket_service, 1164
 socket_acceptor_service, 1192
 stream_socket_service, 1272
puberror
 basic_socket_streambuf, 656

Q

query
 ip::basic_resolver, 969

query_type
 ip::resolver_service, 1007

R

random_access_handle_service
 windows::random_access_handle_service, 1363

raw_protocol
 generic::raw_protocol, 883

raw_socket_service
 raw_socket_service, 1127

rdbuf
 basic_socket_iostream, 617

read, 1128
 read_at, 1137
 read_some
 basic_serial_port, 511
 basic_stream_socket, 711
 buffered_read_stream, 792
 buffered_stream, 801
 buffered_write_stream, 808
 posix::basic_stream_descriptor, 1104
 posix::stream_descriptor_service, 1117
 serial_port_service, 1178
 ssl::stream, 1248
 windows::basic_stream_handle, 1345
 windows::stream_handle_service, 1370

read_some_at
 windows::basic_random_access_handle, 1329
 windows::random_access_handle_service, 1364

read_until, 1146
 receive
 basic_datagram_socket, 341

basic_raw_socket, 424
 basic_seq_packet_socket, 484
 basic_stream_socket, 712
 datagram_socket_service, 847
 raw_socket_service, 1127
 seq_packet_socket_service, 1164
 stream_socket_service, 1272

receive_buffer_size
 basic_datagram_socket, 343
 basic_raw_socket, 426
 basic_seq_packet_socket, 487
 basic_socket, 567
 basic_socket_acceptor, 608
 basic_socket_streambuf, 656
 basic_stream_socket, 715
 socket_base, 1199

receive_from
 basic_datagram_socket, 344
 basic_raw_socket, 427
 datagram_socket_service, 847
 raw_socket_service, 1127

receive_low_watermark
 basic_datagram_socket, 346
 basic_raw_socket, 429
 basic_seq_packet_socket, 487
 basic_socket, 567
 basic_socket_acceptor, 608
 basic_socket_streambuf, 657
 basic_stream_socket, 715
 socket_base, 1200

reference
 buffers_iterator, 817
 ip::basic_resolver_iterator, 980

release
 posix::basic_descriptor, 1085
 posix::basic_stream_descriptor, 1105
 posix::stream_descriptor_service, 1117
 windows::overlapped_ptr, 1356

remote_endpoint
 basic_datagram_socket, 347
 basic_raw_socket, 429
 basic_seq_packet_socket, 488
 basic_socket, 568
 basic_socket_streambuf, 657
 basic_stream_socket, 716
 datagram_socket_service, 847
 raw_socket_service, 1127
 seq_packet_socket_service, 1164
 stream_socket_service, 1272

remove
 basic_signal_set, 526
 signal_set_service, 1184

reserve
 basic_streambuf, 730

reset
 io_service, 917

windows::overlapped_ptr, 1357
 resize
 generic::basic_endpoint, 871
 ip::basic_endpoint, 961
 local::basic_endpoint, 1041
 resolve
 ip::basic_resolver, 970
 ip::resolver_service, 1008
 resolver
 ip::icmp, 992
 ip::tcp, 1017
 ip::udp, 1027
 resolver_service
 ip::resolver_service, 1009
 result_type
 ssl::rfc2818_verification, 1238
 reuse_address
 basic_datagram_socket, 348
 basic_raw_socket, 431
 basic_seq_packet_socket, 489
 basic_socket, 569
 basic_socket_acceptor, 609
 basic_socket_streambuf, 658
 basic_stream_socket, 717
 socket_base, 1200
 rfc2818_verification
 ssl::rfc2818_verification, 1238
 run
 io_service, 918
 run_one
 io_service, 919
 running_in_this_thread
 io_service::strand, 927

S

scope_id
 ip::address_v6, 953
 send
 basic_datagram_socket, 348
 basic_raw_socket, 431
 basic_seq_packet_socket, 489
 basic_stream_socket, 718
 datagram_socket_service, 848
 raw_socket_service, 1127
 seq_packet_socket_service, 1165
 stream_socket_service, 1273
 send_break
 basic_serial_port, 512
 serial_port_service, 1179
 send_buffer_size
 basic_datagram_socket, 351
 basic_raw_socket, 433
 basic_seq_packet_socket, 491
 basic_socket, 570
 basic_socket_acceptor, 609
 basic_socket_streambuf, 659
 basic_stream_socket, 720

socket_base, 1201
 send_low_watermark
 basic_datagram_socket, 351
 basic_raw_socket, 434
 basic_seq_packet_socket, 492
 basic_socket, 570
 basic_socket_acceptor, 610
 basic_socket_streambuf, 660
 basic_stream_socket, 721
 socket_base, 1201
 send_to
 basic_datagram_socket, 352
 basic_raw_socket, 435
 datagram_socket_service, 848
 raw_socket_service, 1128
 seq_packet_protocol
 generic::seq_packet_protocol, 891
 seq_packet_socket_service
 seq_packet_socket_service, 1165
 serial_port, 1165
 serial_port_service
 serial_port_service, 1179
 server
 ssl::stream, 1245
 ssl::stream_base, 1257
 service
 basic_datagram_socket, 354
 basic_deadline_timer, 370
 basic_io_object, 375
 basic_raw_socket, 437
 basic_seq_packet_socket, 492
 basic_serial_port, 513
 basic_signal_set, 527
 basic_socket, 571
 basic_socket_acceptor, 610
 basic_socket_streambuf, 660
 basic_stream_socket, 721
 basic_waitable_timer, 744
 io_service::service, 923
 ip::basic_resolver, 972
 posix::basic_descriptor, 1085
 posix::basic_stream_descriptor, 1106
 windows::basic_handle, 1304
 windows::basic_object_handle, 1316
 windows::basic_random_access_handle, 1331
 windows::basic_stream_handle, 1347
 service_already_exists
 service_already_exists, 1180
 service_name
 ip::basic_resolver_entry, 975
 ip::basic_resolver_query, 987
 service_not_found
 error::addrinfo_errors, 856
 service_type
 basic_datagram_socket, 354
 basic_deadline_timer, 370

basic_io_object, 375
basic_raw_socket, 437
basic_seq_packet_socket, 492
basic_serial_port, 513
basic_signal_set, 527
basic_socket, 571
basic_socket_acceptor, 611
basic_socket_streambuf, 660
basic_stream_socket, 721
basic_waitable_timer, 744
ip::basic_resolver, 973
posix::basic_descriptor, 1086
posix::basic_stream_descriptor, 1106
windows::basic_handle, 1304
windows::basic_object_handle, 1316
windows::basic_random_access_handle, 1331
 windows::basic_stream_handle, 1347

set_default_verify_paths
 ssl::context, 1216

set_option
 basic_datagram_socket, 355
 basic_raw_socket, 437
 basic_seq_packet_socket, 493
 basic_serial_port, 514
 basic_socket, 571
 basic_socket_acceptor, 611
 basic_socket_streambuf, 661
 basic_stream_socket, 722
 datagram_socket_service, 848
 raw_socket_service, 1128
 seq_packet_socket_service, 1165
 serial_port_service, 1179
 socket_acceptor_service, 1192
 stream_socket_service, 1273

set_options
 ssl::context, 1216

set_password_callback
 ssl::context, 1217

set_verify_callback
 ssl::context, 1219
 ssl::stream, 1250

set_verify_depth
 ssl::context, 1220
 ssl::stream, 1251

set_verify_mode
 ssl::context, 1221
 ssl::stream, 1252

setbuf
 basic_socket_streambuf, 662

shut_down
 error::basic_errors, 856

shutdown
 basic_datagram_socket, 356
 basic_raw_socket, 439
 basic_seq_packet_socket, 494
 basic_socket, 573

 basic_socket_streambuf, 662
 basic_stream_socket, 723
 datagram_socket_service, 848
 raw_socket_service, 1128
 seq_packet_socket_service, 1165
 ssl::stream, 1253
 stream_socket_service, 1273

shutdown_both
 basic_datagram_socket, 357
 basic_raw_socket, 440
 basic_seq_packet_socket, 495
 basic_socket, 574
 basic_socket_acceptor, 612
 basic_socket_streambuf, 663
 basic_stream_socket, 724
 socket_base, 1202

shutdown_receive
 basic_datagram_socket, 357
 basic_raw_socket, 440
 basic_seq_packet_socket, 495
 basic_socket, 574
 basic_socket_acceptor, 612
 basic_socket_streambuf, 663
 basic_stream_socket, 724
 socket_base, 1202

shutdown_send
 basic_datagram_socket, 357
 basic_raw_socket, 440
 basic_seq_packet_socket, 495
 basic_socket, 574
 basic_socket_acceptor, 612
 basic_socket_streambuf, 663
 basic_stream_socket, 724
 socket_base, 1202

shutdown_service
 io_service::service, 924

shutdown_type
 basic_datagram_socket, 357
 basic_raw_socket, 440
 basic_seq_packet_socket, 495
 basic_socket, 574
 basic_socket_acceptor, 612
 basic_socket_streambuf, 663
 basic_stream_socket, 724
 socket_base, 1202

signal_set, 1180
signal_set_service
 signal_set_service, 1185

single_dh_use
 ssl::context, 1222
 ssl::context_base, 1237

size
 basic_streambuf, 731
 generic::basic_endpoint, 871
 ip::basic_endpoint, 962
 local::basic_endpoint, 1041

socket
generic::datagram_protocol, 875
generic::raw_protocol, 883
generic::seq_packet_protocol, 891
generic::stream_protocol, 899
ip::icmp, 993
ip::tcp, 1019
ip::udp, 1028
local::datagram_protocol, 1045
local::stream_protocol, 1056
socket_acceptor_service
 socket_acceptor_service, 1192
socket_type_not_supported
 error::addrinfo_errors, 856
software
 serial_port_base::flow_control, 1171
spawn, 1202
ssl
 ssl::stream::impl_struct, 1256
 ssl::verify_client_once, 1257
 ssl::verify_fail_if_no_peer_cert, 1258
 ssl::verify_mode, 1259
 ssl::verify_none, 1259
 ssl::verify_peer, 1259
sslv2
 ssl::context, 1213
 ssl::context_base, 1235
sslv23
 ssl::context, 1213
 ssl::context_base, 1235
sslv23_client
 ssl::context, 1213
 ssl::context_base, 1235
sslv23_server
 ssl::context, 1213
 ssl::context_base, 1235
sslv2_client
 ssl::context, 1213
 ssl::context_base, 1235
sslv2_server
 ssl::context, 1213
 ssl::context_base, 1235
sslv3
 ssl::context, 1213
 ssl::context_base, 1235
sslv3_client
 ssl::context, 1213
 ssl::context_base, 1235
sslv3_server
 ssl::context, 1213
 ssl::context_base, 1235
steady_timer, 1259
stop
 io_service, 920
stop_bits
 serial_port_base::stop_bits, 1173
stopped
 io_service, 920
store
 serial_port_base::baud_rate, 1169
 serial_port_base::character_size, 1170
 serial_port_base::flow_control, 1171
 serial_port_base::parity, 1172
 serial_port_base::stop_bits, 1173
strand, 1262
 io_service::strand, 927
stream
 ssl::stream, 1254
stream_descriptor_service
 posix::stream_descriptor_service, 1117
stream_handle_service
 windows::stream_handle_service, 1370
stream_protocol
 generic::stream_protocol, 903
stream_socket_service
 stream_socket_service, 1273
streambuf, 1273
subtract
 time_traits< boost::posix_time::ptime >, 1284
sync
 basic_socket_streambuf, 664
system_category, 1275
system_error
 system_error, 1276
system_timer, 1277
T
thread
 thread, 1281
time_point
 basic_waitable_timer, 744
 waitable_timer_service, 1293
time_type
 basic_deadline_timer, 370
 basic_socket_iostream, 617
 basic_socket_streambuf, 664
 deadline_timer_service, 855
 time_traits< boost::posix_time::ptime >, 1284
timed_out
 error::basic_errors, 856
timer_handler
 basic_socket_streambuf, 664
tlsv1
 ssl::context, 1213
 ssl::context_base, 1235
tlsv11
 ssl::context, 1213
 ssl::context_base, 1235
tlsv11_client
 ssl::context, 1213
 ssl::context_base, 1235
tlsv11_server
 ssl::context, 1213

ssl::context_base, 1235
 tlsv12
 ssl::context, 1213
 ssl::context_base, 1235
 tlsv12_client
 ssl::context, 1213
 ssl::context_base, 1235
 tlsv12_server
 ssl::context, 1213
 ssl::context_base, 1235
 tlsv1_client
 ssl::context, 1213
 ssl::context_base, 1235
 tlsv1_server
 ssl::context, 1213
 ssl::context_base, 1235
 to_bytes
 ip::address_v4, 944
 ip::address_v6, 953
 to_posix_duration
 time_traits< boost::posix_time::ptime >, 1284
 to_string
 ip::address, 936
 ip::address_v4, 945
 ip::address_v6, 954
 to_ulong
 ip::address_v4, 945
 to_v4
 ip::address, 937
 ip::address_v6, 954
 to_v6
 ip::address, 937
 to_wait_duration
 wait_traits, 1289
 traits_type
 basic_deadline_timer, 371
 basic_waitable_timer, 744
 deadline_timer_service, 855
 waitable_timer_service, 1293
 transfer_all, 1284
 transfer_at_least, 1285
 transfer_exactly, 1285
 try_again
 error::basic_errors, 856
 two
 serial_port_base::stop_bits, 1173
 type
 async_result, 283
 generic::datagram_protocol, 879
 generic::raw_protocol, 887
 generic::seq_packet_protocol, 895
 generic::stream_protocol, 904
 handler_type, 904
 ip::icmp, 997
 ip::tcp, 1023
 ip::udp, 1032
 local::datagram_protocol, 1049
 local::stream_protocol, 1060
 serial_port_base::flow_control, 1171
 serial_port_base::parity, 1172
 serial_port_base::stop_bits, 1173

U

underflow
 basic_socket_streambuf, 664
 basic_streambuf, 731
 unspecified_bool_true
 error_code, 865
 unspecified_bool_type
 error_code, 865
 use_certificate
 ssl::context, 1222
 use_certificate_chain
 ssl::context, 1223
 use_certificate_chain_file
 ssl::context, 1224
 use_certificate_file
 ssl::context, 1225
 use_future, 1286
 use_future_t
 use_future_t, 1288
 use_private_key
 ssl::context, 1227
 use_private_key_file
 ssl::context, 1228
 use_rsa_private_key
 ssl::context, 1229
 use_rsa_private_key_file
 ssl::context, 1230
 use_service, 1288
 io_service, 920
 use_tmp_dh
 ssl::context, 1231
 use_tmp_dh_file
 ssl::context, 1232

V

v4
 ip::icmp, 997
 ip::tcp, 1023
 ip::udp, 1032
 v4_compatible
 ip::address_v6, 954
 v4_mapped
 ip::address_v6, 954
 ip::basic_resolver_query, 988
 ip::resolver_query_base, 1002
 v6
 ip::icmp, 997
 ip::tcp, 1023
 ip::udp, 1032
 value
 error_code, 866

is_match_condition, 1034
is_read_buffered, 1035
is_write_buffered, 1035
serial_port_base::baud_rate, 1169
serial_port_base::character_size, 1170
serial_port_base::flow_control, 1171
serial_port_base::parity, 1172
serial_port_base::stop_bits, 1174
value_type
 buffers_iterator, 818
 const_buffers_1, 833
 ip::basic_resolver_iterator, 981
 mutable_buffers_1, 1065
 null_buffers, 1067
verify_context
 ssl::verify_context, 1258

W
wait
 basic_deadline_timer, 371
 basic_waitable_timer, 745
 deadline_timer_service, 856
 waitable_timer_service, 1294
 windows::basic_object_handle, 1316
 windows::object_handle_service, 1354
waitable_timer_service
 waitable_timer_service, 1294
what
 system_error, 1277
windows::object_handle, 1349
windows::random_access_handle, 1357
windows::stream_handle, 1364
work
 io_service::work, 929
would_block
 error::basic_errors, 856
wrap
 io_service, 921
 io_service::strand, 927
write, 1371
write_at, 1379
write_some
 basic_serial_port, 515
 basic_stream_socket, 725
 buffered_read_stream, 793
 buffered_stream, 801
 buffered_write_stream, 809
 posix::basic_stream_descriptor, 1106
 posix::stream_descriptor_service, 1118
 serial_port_service, 1179
 ssl::stream, 1254
 windows::basic_stream_handle, 1347
 windows::stream_handle_service, 1371
write_some_at
 windows::basic_random_access_handle, 1332
 windows::random_access_handle_service, 1364

Y
yield_context, 1389