

asio C++ library

1.12.0

Reference Manual

Copyright © 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017 Christopher M. Kohlhoff

Contents

1 Overview	81
1.1 Rationale	82
1.2 Core Concepts and Functionality	82
1.2.1 Basic Asio Anatomy	83
1.2.2 The Proactor Design Pattern: Concurrency Without Threads	85
1.2.3 Threads and Asio	88
1.2.4 Strands: Use Threads Without Explicit Locking	89
1.2.5 Buffers	90
1.2.6 Streams, Short Reads and Short Writes	92
1.2.7 Reactor-Style Operations	93
1.2.8 Line-Based Operations	93
1.2.9 Custom Memory Allocation	95
1.2.10 Handler Tracking	96
1.2.11 Concurrency Hints	99
1.2.12 Stackless Coroutines	100
1.2.13 Stackful Coroutines	101
1.3 Networking	102
1.3.1 TCP, UDP and ICMP	102
1.3.2 Support for Other Protocols	104
1.3.3 Socket Iostreams	105
1.3.4 The BSD Socket API and Asio	106
1.4 Timers	108
1.5 Serial Ports	109
1.6 Signal Handling	110
1.7 POSIX-Specific Functionality	110
1.7.1 UNIX Domain Sockets	110
1.7.2 Stream-Oriented File Descriptors	111
1.7.3 Fork	112
1.8 Windows-Specific Functionality	112
1.8.1 Stream-Oriented HANDLEs	112
1.8.2 Random-Access HANDLEs	113
1.8.3 Object HANDLEs	113
1.9 SSL	114
1.10 C++ 2011 Support	115
1.10.1 System Errors and Error Codes	116
1.10.2 Movable I/O Objects	116
1.10.3 Movable Handlers	117

1.10.4	Variadic Templates	117
1.10.5	Array Container	117
1.10.6	Atomics	118
1.10.7	Shared Pointers	118
1.10.8	Chrono	118
1.10.9	Futures	118
1.11	Platform-Specific Implementation Notes	118
2	Using Asio	124
3	Tutorial	129
3.1	Timer.1 - Using a timer synchronously	129
3.1.1	Source listing for Timer.1	130
3.2	Timer.2 - Using a timer asynchronously	131
3.2.1	Source listing for Timer.2	131
3.3	Timer.3 - Binding arguments to a handler	132
3.3.1	Source listing for Timer.3	133
3.4	Timer.4 - Using a member function as a handler	134
3.4.1	Source listing for Timer.4	136
3.5	Timer.5 - Synchronising handlers in multithreaded programs	137
3.5.1	Source listing for Timer.5	138
3.6	Daytime.1 - A synchronous TCP daytime client	140
3.6.1	Source listing for Daytime.1	141
3.7	Daytime.2 - A synchronous TCP daytime server	142
3.7.1	Source listing for Daytime.2	143
3.8	Daytime.3 - An asynchronous TCP daytime server	144
3.8.1	Source listing for Daytime.3	146
3.9	Daytime.4 - A synchronous UDP daytime client	148
3.9.1	Source listing for Daytime.4	149
3.10	Daytime.5 - A synchronous UDP daytime server	150
3.10.1	Source listing for Daytime.5	151
3.11	Daytime.6 - An asynchronous UDP daytime server	152
3.11.1	Source listing for Daytime.6	154
3.12	Daytime.7 - A combined TCP/UDP asynchronous server	155
3.12.1	Source listing for Daytime.7	158
3.13	boost::bind	160
4	Examples	161
4.1	C++03 Examples	161
4.2	C++11 Examples	167

5 Reference	170
5.1 Requirements on asynchronous operations	172
5.1.1 General asynchronous operation concepts	172
5.1.2 Completion tokens and handlers	172
5.1.3 Automatic deduction of initiating function return type	173
5.1.4 Production of initiating function return value	173
5.1.5 Lifetime of initiating function arguments	173
5.1.6 Non-blocking requirements on initiating functions	174
5.1.7 Associated executor	174
5.1.8 I/O executor	174
5.1.9 Completion handler executor	174
5.1.10 Outstanding work	174
5.1.11 Allocation of intermediate storage	174
5.1.12 Execution of completion handler on completion of asynchronous operation	175
5.1.13 Completion handlers and exceptions	175
5.2 Requirements on read and write operations	175
5.3 Requirements on synchronous socket operations	175
5.4 Requirements on asynchronous socket operations	176
5.5 Acceptable protocol requirements	176
5.6 Accept handler requirements	177
5.7 Buffer-oriented asynchronous random-access read device requirements	178
5.8 Buffer-oriented asynchronous random-access write device requirements	179
5.9 Buffer-oriented asynchronous read stream requirements	180
5.10 Buffer-oriented asynchronous write stream requirements	180
5.11 Buffered handshake handler requirements	181
5.12 Completion condition requirements	182
5.13 Completion handler requirements	182
5.14 Connect condition requirements	183
5.15 Connect handler requirements	184
5.16 Constant buffer sequence requirements	185
5.17 Dynamic buffer requirements	186
5.18 Endpoint requirements	188
5.19 Endpoint sequence requirements	189
5.20 Execution context requirements	189
5.21 Executor requirements	190
5.22 Gettable serial port option requirements	193
5.23 Gettable socket option requirements	194
5.24 Handlers	195
5.25 SSL handshake handler requirements	196

5.26	Internet protocol requirements	197
5.27	I/O control command requirements	197
5.28	I/O object service requirements	197
5.29	Iterator connect handler requirements	198
5.30	Move accept handler requirements	199
5.31	Mutable buffer sequence requirements	200
5.32	Proto-allocator requirements	201
5.33	Protocol requirements	201
5.34	Range connect handler requirements	202
5.35	Read handler requirements	203
5.36	Resolve handler requirements	204
5.37	Service requirements	205
5.38	Settable serial port option requirements	206
5.39	Settable socket option requirements	206
5.40	SSL shutdown handler requirements	207
5.41	Signal handler requirements	208
5.42	Buffer-oriented synchronous random-access read device requirements	209
5.43	Buffer-oriented synchronous random-access write device requirements	210
5.44	Buffer-oriented synchronous read stream requirements	211
5.45	Buffer-oriented synchronous write stream requirements	212
5.46	Time traits requirements	212
5.47	Wait handler requirements	213
5.48	Wait traits requirements	214
5.49	Write handler requirements	215
5.50	add_service	216
5.51	asio_handler_allocate	217
5.52	asio_handler_deallocate	217
5.53	asio_handler_invoke	218
5.53.1	asio_handler_invoke (1 of 2 overloads)	219
5.53.2	asio_handler_invoke (2 of 2 overloads)	219
5.54	asio_handler_is_continuation	219
5.55	associated_allocator	219
5.55.1	associated_allocator::get	220
5.55.2	associated_allocator::type	220
5.56	associated_executor	221
5.56.1	associated_executor::get	221
5.56.2	associated_executor::type	222
5.57	async_completion	222
5.57.1	async_completion::async_completion	223

5.57.2	async_completion::completion_handler	223
5.57.3	async_completion::completion_handler_type	223
5.57.4	async_completion::result	224
5.58	async_connect	224
5.58.1	async_connect (1 of 6 overloads)	225
5.58.2	async_connect (2 of 6 overloads)	226
5.58.3	async_connect (3 of 6 overloads)	227
5.58.4	async_connect (4 of 6 overloads)	228
5.58.5	async_connect (5 of 6 overloads)	230
5.58.6	async_connect (6 of 6 overloads)	231
5.59	async_read	233
5.59.1	async_read (1 of 6 overloads)	234
5.59.2	async_read (2 of 6 overloads)	235
5.59.3	async_read (3 of 6 overloads)	237
5.59.4	async_read (4 of 6 overloads)	238
5.59.5	async_read (5 of 6 overloads)	239
5.59.6	async_read (6 of 6 overloads)	240
5.60	async_read_at	241
5.60.1	async_read_at (1 of 4 overloads)	241
5.60.2	async_read_at (2 of 4 overloads)	243
5.60.3	async_read_at (3 of 4 overloads)	244
5.60.4	async_read_at (4 of 4 overloads)	245
5.61	async_read_until	246
5.61.1	async_read_until (1 of 8 overloads)	248
5.61.2	async_read_until (2 of 8 overloads)	249
5.61.3	async_read_until (3 of 8 overloads)	251
5.61.4	async_read_until (4 of 8 overloads)	252
5.61.5	async_read_until (5 of 8 overloads)	254
5.61.6	async_read_until (6 of 8 overloads)	256
5.61.7	async_read_until (7 of 8 overloads)	257
5.61.8	async_read_until (8 of 8 overloads)	259
5.62	async_result	261
5.62.1	async_result::async_result	262
5.62.2	async_result::completion_handler_type	262
5.62.3	async_result::get	262
5.62.4	async_result::return_type	262
5.63	async_result< Handler >	263
5.63.1	async_result< Handler >::async_result	263
5.63.2	async_result< Handler >::get	263

5.63.3	async_result< Handler >::type	264
5.64	async_result< std::packaged_task< Result(Args...)>, Signature >	264
5.64.1	async_result< std::packaged_task< Result(Args...)>, Signature >::async_result	264
5.64.2	async_result< std::packaged_task< Result(Args...)>, Signature >::completion_handler_type	265
5.64.3	async_result< std::packaged_task< Result(Args...)>, Signature >::get	265
5.64.4	async_result< std::packaged_task< Result(Args...)>, Signature >::return_type	265
5.65	async_write	265
5.65.1	async_write (1 of 6 overloads)	266
5.65.2	async_write (2 of 6 overloads)	267
5.65.3	async_write (3 of 6 overloads)	269
5.65.4	async_write (4 of 6 overloads)	270
5.65.5	async_write (5 of 6 overloads)	271
5.65.6	async_write (6 of 6 overloads)	271
5.66	async_write_at	272
5.66.1	async_write_at (1 of 4 overloads)	273
5.66.2	async_write_at (2 of 4 overloads)	274
5.66.3	async_write_at (3 of 4 overloads)	276
5.66.4	async_write_at (4 of 4 overloads)	276
5.67	bad_executor	278
5.67.1	bad_executor::bad_executor	278
5.67.2	bad_executor::what	278
5.68	basic_datagram_socket	278
5.68.1	basic_datagram_socket::assign	282
5.68.1.1	basic_datagram_socket::assign (1 of 2 overloads)	282
5.68.1.2	basic_datagram_socket::assign (2 of 2 overloads)	282
5.68.2	basic_datagram_socket::async_connect	283
5.68.3	basic_datagram_socket::async_receive	283
5.68.3.1	basic_datagram_socket::async_receive (1 of 2 overloads)	284
5.68.3.2	basic_datagram_socket::async_receive (2 of 2 overloads)	285
5.68.4	basic_datagram_socket::async_receive_from	285
5.68.4.1	basic_datagram_socket::async_receive_from (1 of 2 overloads)	286
5.68.4.2	basic_datagram_socket::async_receive_from (2 of 2 overloads)	286
5.68.5	basic_datagram_socket::async_send	287
5.68.5.1	basic_datagram_socket::async_send (1 of 2 overloads)	287
5.68.5.2	basic_datagram_socket::async_send (2 of 2 overloads)	288
5.68.6	basic_datagram_socket::async_send_to	289
5.68.6.1	basic_datagram_socket::async_send_to (1 of 2 overloads)	289
5.68.6.2	basic_datagram_socket::async_send_to (2 of 2 overloads)	290
5.68.7	basic_datagram_socket::async_wait	290

5.68.8	<code>basic_datagram_socket::at_mark</code>	291
5.68.8.1	<code>basic_datagram_socket::at_mark</code> (1 of 2 overloads)	291
5.68.8.2	<code>basic_datagram_socket::at_mark</code> (2 of 2 overloads)	292
5.68.9	<code>basic_datagram_socket::available</code>	292
5.68.9.1	<code>basic_datagram_socket::available</code> (1 of 2 overloads)	292
5.68.9.2	<code>basic_datagram_socket::available</code> (2 of 2 overloads)	293
5.68.10	<code>basic_datagram_socket::basic_datagram_socket</code>	293
5.68.10.1	<code>basic_datagram_socket::basic_datagram_socket</code> (1 of 6 overloads)	294
5.68.10.2	<code>basic_datagram_socket::basic_datagram_socket</code> (2 of 6 overloads)	294
5.68.10.3	<code>basic_datagram_socket::basic_datagram_socket</code> (3 of 6 overloads)	294
5.68.10.4	<code>basic_datagram_socket::basic_datagram_socket</code> (4 of 6 overloads)	295
5.68.10.5	<code>basic_datagram_socket::basic_datagram_socket</code> (5 of 6 overloads)	295
5.68.10.6	<code>basic_datagram_socket::basic_datagram_socket</code> (6 of 6 overloads)	295
5.68.11	<code>basic_datagram_socket::bind</code>	296
5.68.11.1	<code>basic_datagram_socket::bind</code> (1 of 2 overloads)	296
5.68.11.2	<code>basic_datagram_socket::bind</code> (2 of 2 overloads)	297
5.68.12	<code>basic_datagram_socket::broadcast</code>	297
5.68.13	<code>basic_datagram_socket::bytes_readable</code>	298
5.68.14	<code>basic_datagram_socket::cancel</code>	298
5.68.14.1	<code>basic_datagram_socket::cancel</code> (1 of 2 overloads)	298
5.68.14.2	<code>basic_datagram_socket::cancel</code> (2 of 2 overloads)	299
5.68.15	<code>basic_datagram_socket::close</code>	300
5.68.15.1	<code>basic_datagram_socket::close</code> (1 of 2 overloads)	300
5.68.15.2	<code>basic_datagram_socket::close</code> (2 of 2 overloads)	300
5.68.16	<code>basic_datagram_socket::connect</code>	301
5.68.16.1	<code>basic_datagram_socket::connect</code> (1 of 2 overloads)	301
5.68.16.2	<code>basic_datagram_socket::connect</code> (2 of 2 overloads)	302
5.68.17	<code>basic_datagram_socket::debug</code>	302
5.68.18	<code>basic_datagram_socket::do_not_route</code>	303
5.68.19	<code>basic_datagram_socket::enable_connection_aborted</code>	303
5.68.20	<code>basic_datagram_socket::endpoint_type</code>	304
5.68.21	<code>basic_datagram_socket::executor_type</code>	304
5.68.22	<code>basic_datagram_socket::get_executor</code>	305
5.68.23	<code>basic_datagram_socket::get_io_context</code>	305
5.68.24	<code>basic_datagram_socket::get_io_service</code>	305
5.68.25	<code>basic_datagram_socket::get_option</code>	306
5.68.25.1	<code>basic_datagram_socket::get_option</code> (1 of 2 overloads)	306
5.68.25.2	<code>basic_datagram_socket::get_option</code> (2 of 2 overloads)	307
5.68.26	<code>basic_datagram_socket::io_control</code>	307

5.68.26.1	<code>basic_datagram_socket::io_control</code> (1 of 2 overloads)	308
5.68.26.2	<code>basic_datagram_socket::io_control</code> (2 of 2 overloads)	308
5.68.27	<code>basic_datagram_socket::is_open</code>	309
5.68.28	<code>basic_datagram_socket::keep_alive</code>	309
5.68.29	<code>basic_datagram_socket::linger</code>	310
5.68.30	<code>basic_datagram_socket::local_endpoint</code>	310
5.68.30.1	<code>basic_datagram_socket::local_endpoint</code> (1 of 2 overloads)	310
5.68.30.2	<code>basic_datagram_socket::local_endpoint</code> (2 of 2 overloads)	311
5.68.31	<code>basic_datagram_socket::lowest_layer</code>	311
5.68.31.1	<code>basic_datagram_socket::lowest_layer</code> (1 of 2 overloads)	312
5.68.31.2	<code>basic_datagram_socket::lowest_layer</code> (2 of 2 overloads)	312
5.68.32	<code>basic_datagram_socket::lowest_layer_type</code>	312
5.68.33	<code>basic_datagram_socket::max_connections</code>	316
5.68.34	<code>basic_datagram_socket::max_listen_connections</code>	316
5.68.35	<code>basic_datagram_socket::message_do_not_route</code>	316
5.68.36	<code>basic_datagram_socket::message_end_of_record</code>	316
5.68.37	<code>basic_datagram_socket::message_flags</code>	316
5.68.38	<code>basic_datagram_socket::message_out_of_band</code>	316
5.68.39	<code>basic_datagram_socket::message_peek</code>	317
5.68.40	<code>basic_datagram_socket::native_handle</code>	317
5.68.41	<code>basic_datagram_socket::native_handle_type</code>	317
5.68.42	<code>basic_datagram_socket::native_non_blocking</code>	317
5.68.42.1	<code>basic_datagram_socket::native_non_blocking</code> (1 of 3 overloads)	317
5.68.42.2	<code>basic_datagram_socket::native_non_blocking</code> (2 of 3 overloads)	319
5.68.42.3	<code>basic_datagram_socket::native_non_blocking</code> (3 of 3 overloads)	320
5.68.43	<code>basic_datagram_socket::non_blocking</code>	322
5.68.43.1	<code>basic_datagram_socket::non_blocking</code> (1 of 3 overloads)	322
5.68.43.2	<code>basic_datagram_socket::non_blocking</code> (2 of 3 overloads)	323
5.68.43.3	<code>basic_datagram_socket::non_blocking</code> (3 of 3 overloads)	323
5.68.44	<code>basic_datagram_socket::open</code>	324
5.68.44.1	<code>basic_datagram_socket::open</code> (1 of 2 overloads)	324
5.68.44.2	<code>basic_datagram_socket::open</code> (2 of 2 overloads)	324
5.68.45	<code>basic_datagram_socket::operator=</code>	325
5.68.45.1	<code>basic_datagram_socket::operator=</code> (1 of 2 overloads)	325
5.68.45.2	<code>basic_datagram_socket::operator=</code> (2 of 2 overloads)	325
5.68.46	<code>basic_datagram_socket::out_of_band_inline</code>	326
5.68.47	<code>basic_datagram_socket::protocol_type</code>	326
5.68.48	<code>basic_datagram_socket::receive</code>	327
5.68.48.1	<code>basic_datagram_socket::receive</code> (1 of 3 overloads)	327

5.68.48.2	basic_datagram_socket::receive (2 of 3 overloads)	328
5.68.48.3	basic_datagram_socket::receive (3 of 3 overloads)	328
5.68.49	basic_datagram_socket::receive_buffer_size	329
5.68.50	basic_datagram_socket::receive_from	330
5.68.50.1	basic_datagram_socket::receive_from (1 of 3 overloads)	330
5.68.50.2	basic_datagram_socket::receive_from (2 of 3 overloads)	331
5.68.50.3	basic_datagram_socket::receive_from (3 of 3 overloads)	331
5.68.51	basic_datagram_socket::receive_low_watermark	332
5.68.52	basic_datagram_socket::release	332
5.68.52.1	basic_datagram_socket::release (1 of 2 overloads)	333
5.68.52.2	basic_datagram_socket::release (2 of 2 overloads)	333
5.68.53	basic_datagram_socket::remote_endpoint	333
5.68.53.1	basic_datagram_socket::remote_endpoint (1 of 2 overloads)	334
5.68.53.2	basic_datagram_socket::remote_endpoint (2 of 2 overloads)	334
5.68.54	basic_datagram_socket::reuse_address	335
5.68.55	basic_datagram_socket::send	335
5.68.55.1	basic_datagram_socket::send (1 of 3 overloads)	336
5.68.55.2	basic_datagram_socket::send (2 of 3 overloads)	336
5.68.55.3	basic_datagram_socket::send (3 of 3 overloads)	337
5.68.56	basic_datagram_socket::send_buffer_size	337
5.68.57	basic_datagram_socket::send_low_watermark	338
5.68.58	basic_datagram_socket::send_to	339
5.68.58.1	basic_datagram_socket::send_to (1 of 3 overloads)	339
5.68.58.2	basic_datagram_socket::send_to (2 of 3 overloads)	340
5.68.58.3	basic_datagram_socket::send_to (3 of 3 overloads)	340
5.68.59	basic_datagram_socket::set_option	341
5.68.59.1	basic_datagram_socket::set_option (1 of 2 overloads)	341
5.68.59.2	basic_datagram_socket::set_option (2 of 2 overloads)	342
5.68.60	basic_datagram_socket::shutdown	342
5.68.60.1	basic_datagram_socket::shutdown (1 of 2 overloads)	343
5.68.60.2	basic_datagram_socket::shutdown (2 of 2 overloads)	343
5.68.61	basic_datagram_socket::shutdown_type	344
5.68.62	basic_datagram_socket::wait	344
5.68.62.1	basic_datagram_socket::wait (1 of 2 overloads)	344
5.68.62.2	basic_datagram_socket::wait (2 of 2 overloads)	345
5.68.63	basic_datagram_socket::wait_type	345
5.68.64	basic_datagram_socket::~basic_datagram_socket	345
5.69	basic_deadline_timer	346
5.69.1	basic_deadline_timer::async_wait	348

5.69.2	<code>basic_deadline_timer::basic_deadline_timer</code>	349
5.69.2.1	<code>basic_deadline_timer::basic_deadline_timer</code> (1 of 4 overloads)	349
5.69.2.2	<code>basic_deadline_timer::basic_deadline_timer</code> (2 of 4 overloads)	349
5.69.2.3	<code>basic_deadline_timer::basic_deadline_timer</code> (3 of 4 overloads)	350
5.69.2.4	<code>basic_deadline_timer::basic_deadline_timer</code> (4 of 4 overloads)	350
5.69.3	<code>basic_deadline_timer::cancel</code>	350
5.69.3.1	<code>basic_deadline_timer::cancel</code> (1 of 2 overloads)	350
5.69.3.2	<code>basic_deadline_timer::cancel</code> (2 of 2 overloads)	351
5.69.4	<code>basic_deadline_timer::cancel_one</code>	352
5.69.4.1	<code>basic_deadline_timer::cancel_one</code> (1 of 2 overloads)	352
5.69.4.2	<code>basic_deadline_timer::cancel_one</code> (2 of 2 overloads)	352
5.69.5	<code>basic_deadline_timer::duration_type</code>	353
5.69.6	<code>basic_deadline_timer::executor_type</code>	353
5.69.7	<code>basic_deadline_timer::expires_at</code>	354
5.69.7.1	<code>basic_deadline_timer::expires_at</code> (1 of 3 overloads)	354
5.69.7.2	<code>basic_deadline_timer::expires_at</code> (2 of 3 overloads)	354
5.69.7.3	<code>basic_deadline_timer::expires_at</code> (3 of 3 overloads)	355
5.69.8	<code>basic_deadline_timer::expires_from_now</code>	356
5.69.8.1	<code>basic_deadline_timer::expires_from_now</code> (1 of 3 overloads)	356
5.69.8.2	<code>basic_deadline_timer::expires_from_now</code> (2 of 3 overloads)	356
5.69.8.3	<code>basic_deadline_timer::expires_from_now</code> (3 of 3 overloads)	357
5.69.9	<code>basic_deadline_timer::get_executor</code>	357
5.69.10	<code>basic_deadline_timer::get_io_context</code>	357
5.69.11	<code>basic_deadline_timer::get_io_service</code>	358
5.69.12	<code>basic_deadline_timer::operator=</code>	358
5.69.13	<code>basic_deadline_timer::time_type</code>	358
5.69.14	<code>basic_deadline_timer::traits_type</code>	358
5.69.15	<code>basic_deadline_timer::wait</code>	359
5.69.15.1	<code>basic_deadline_timer::wait</code> (1 of 2 overloads)	359
5.69.15.2	<code>basic_deadline_timer::wait</code> (2 of 2 overloads)	359
5.69.16	<code>basic_deadline_timer::~basic_deadline_timer</code>	359
5.70	<code>basic_io_object</code>	359
5.70.1	<code>basic_io_object::basic_io_object</code>	360
5.70.1.1	<code>basic_io_object::basic_io_object</code> (1 of 3 overloads)	361
5.70.1.2	<code>basic_io_object::basic_io_object</code> (2 of 3 overloads)	361
5.70.1.3	<code>basic_io_object::basic_io_object</code> (3 of 3 overloads)	361
5.70.2	<code>basic_io_object::executor_type</code>	362
5.70.3	<code>basic_io_object::get_executor</code>	362
5.70.4	<code>basic_io_object::get_implementation</code>	362

5.70.4.1	<code>basic_io_object::get_implementation</code> (1 of 2 overloads)	363
5.70.4.2	<code>basic_io_object::get_implementation</code> (2 of 2 overloads)	363
5.70.5	<code>basic_io_object::get_io_context</code>	363
5.70.6	<code>basic_io_object::get_io_service</code>	363
5.70.7	<code>basic_io_object::get_service</code>	363
5.70.7.1	<code>basic_io_object::get_service</code> (1 of 2 overloads)	364
5.70.7.2	<code>basic_io_object::get_service</code> (2 of 2 overloads)	364
5.70.8	<code>basic_io_object::implementation_type</code>	364
5.70.9	<code>basic_io_object::operator=</code>	364
5.70.10	<code>basic_io_object::service_type</code>	364
5.70.11	<code>basic_io_object::~basic_io_object</code>	365
5.71	<code>basic_raw_socket</code>	365
5.71.1	<code>basic_raw_socket::assign</code>	368
5.71.1.1	<code>basic_raw_socket::assign</code> (1 of 2 overloads)	369
5.71.1.2	<code>basic_raw_socket::assign</code> (2 of 2 overloads)	369
5.71.2	<code>basic_raw_socket::async_connect</code>	369
5.71.3	<code>basic_raw_socket::async_receive</code>	370
5.71.3.1	<code>basic_raw_socket::async_receive</code> (1 of 2 overloads)	370
5.71.3.2	<code>basic_raw_socket::async_receive</code> (2 of 2 overloads)	371
5.71.4	<code>basic_raw_socket::async_receive_from</code>	372
5.71.4.1	<code>basic_raw_socket::async_receive_from</code> (1 of 2 overloads)	372
5.71.4.2	<code>basic_raw_socket::async_receive_from</code> (2 of 2 overloads)	373
5.71.5	<code>basic_raw_socket::async_send</code>	373
5.71.5.1	<code>basic_raw_socket::async_send</code> (1 of 2 overloads)	374
5.71.5.2	<code>basic_raw_socket::async_send</code> (2 of 2 overloads)	375
5.71.6	<code>basic_raw_socket::async_send_to</code>	375
5.71.6.1	<code>basic_raw_socket::async_send_to</code> (1 of 2 overloads)	376
5.71.6.2	<code>basic_raw_socket::async_send_to</code> (2 of 2 overloads)	376
5.71.7	<code>basic_raw_socket::async_wait</code>	377
5.71.8	<code>basic_raw_socket::at_mark</code>	378
5.71.8.1	<code>basic_raw_socket::at_mark</code> (1 of 2 overloads)	378
5.71.8.2	<code>basic_raw_socket::at_mark</code> (2 of 2 overloads)	378
5.71.9	<code>basic_raw_socket::available</code>	379
5.71.9.1	<code>basic_raw_socket::available</code> (1 of 2 overloads)	379
5.71.9.2	<code>basic_raw_socket::available</code> (2 of 2 overloads)	379
5.71.10	<code>basic_raw_socket::basic_raw_socket</code>	380
5.71.10.1	<code>basic_raw_socket::basic_raw_socket</code> (1 of 6 overloads)	380
5.71.10.2	<code>basic_raw_socket::basic_raw_socket</code> (2 of 6 overloads)	381
5.71.10.3	<code>basic_raw_socket::basic_raw_socket</code> (3 of 6 overloads)	381

5.71.10.4	basic_raw_socket::basic_raw_socket (4 of 6 overloads)	381
5.71.10.5	basic_raw_socket::basic_raw_socket (5 of 6 overloads)	382
5.71.10.6	basic_raw_socket::basic_raw_socket (6 of 6 overloads)	382
5.71.11	basic_raw_socket::bind	383
5.71.11.1	basic_raw_socket::bind (1 of 2 overloads)	383
5.71.11.2	basic_raw_socket::bind (2 of 2 overloads)	383
5.71.12	basic_raw_socket::broadcast	384
5.71.13	basic_raw_socket::bytes_readable	384
5.71.14	basic_raw_socket::cancel	385
5.71.14.1	basic_raw_socket::cancel (1 of 2 overloads)	385
5.71.14.2	basic_raw_socket::cancel (2 of 2 overloads)	386
5.71.15	basic_raw_socket::close	386
5.71.15.1	basic_raw_socket::close (1 of 2 overloads)	386
5.71.15.2	basic_raw_socket::close (2 of 2 overloads)	387
5.71.16	basic_raw_socket::connect	387
5.71.16.1	basic_raw_socket::connect (1 of 2 overloads)	388
5.71.16.2	basic_raw_socket::connect (2 of 2 overloads)	388
5.71.17	basic_raw_socket::debug	389
5.71.18	basic_raw_socket::do_not_route	389
5.71.19	basic_raw_socket::enable_connection_aborted	390
5.71.20	basic_raw_socket::endpoint_type	391
5.71.21	basic_raw_socket::executor_type	391
5.71.22	basic_raw_socket::get_executor	392
5.71.23	basic_raw_socket::get_io_context	392
5.71.24	basic_raw_socket::get_io_service	392
5.71.25	basic_raw_socket::get_option	392
5.71.25.1	basic_raw_socket::get_option (1 of 2 overloads)	393
5.71.25.2	basic_raw_socket::get_option (2 of 2 overloads)	393
5.71.26	basic_raw_socket::io_control	394
5.71.26.1	basic_raw_socket::io_control (1 of 2 overloads)	394
5.71.26.2	basic_raw_socket::io_control (2 of 2 overloads)	395
5.71.27	basic_raw_socket::is_open	395
5.71.28	basic_raw_socket::keep_alive	396
5.71.29	basic_raw_socket::linger	396
5.71.30	basic_raw_socket::local_endpoint	397
5.71.30.1	basic_raw_socket::local_endpoint (1 of 2 overloads)	397
5.71.30.2	basic_raw_socket::local_endpoint (2 of 2 overloads)	397
5.71.31	basic_raw_socket::lowest_layer	398
5.71.31.1	basic_raw_socket::lowest_layer (1 of 2 overloads)	398

5.71.31.2	<code>basic_raw_socket::lowest_layer</code> (2 of 2 overloads)	398
5.71.32	<code>basic_raw_socket::lowest_layer_type</code>	399
5.71.33	<code>basic_raw_socket::max_connections</code>	402
5.71.34	<code>basic_raw_socket::max_listen_connections</code>	402
5.71.35	<code>basic_raw_socket::message_do_not_route</code>	402
5.71.36	<code>basic_raw_socket::message_end_of_record</code>	402
5.71.37	<code>basic_raw_socket::message_flags</code>	403
5.71.38	<code>basic_raw_socket::message_out_of_band</code>	403
5.71.39	<code>basic_raw_socket::message_peek</code>	403
5.71.40	<code>basic_raw_socket::native_handle</code>	403
5.71.41	<code>basic_raw_socket::native_handle_type</code>	403
5.71.42	<code>basic_raw_socket::native_non_blocking</code>	404
5.71.42.1	<code>basic_raw_socket::native_non_blocking</code> (1 of 3 overloads)	404
5.71.42.2	<code>basic_raw_socket::native_non_blocking</code> (2 of 3 overloads)	405
5.71.42.3	<code>basic_raw_socket::native_non_blocking</code> (3 of 3 overloads)	407
5.71.43	<code>basic_raw_socket::non_blocking</code>	409
5.71.43.1	<code>basic_raw_socket::non_blocking</code> (1 of 3 overloads)	409
5.71.43.2	<code>basic_raw_socket::non_blocking</code> (2 of 3 overloads)	409
5.71.43.3	<code>basic_raw_socket::non_blocking</code> (3 of 3 overloads)	410
5.71.44	<code>basic_raw_socket::open</code>	410
5.71.44.1	<code>basic_raw_socket::open</code> (1 of 2 overloads)	410
5.71.44.2	<code>basic_raw_socket::open</code> (2 of 2 overloads)	411
5.71.45	<code>basic_raw_socket::operator=</code>	411
5.71.45.1	<code>basic_raw_socket::operator=</code> (1 of 2 overloads)	412
5.71.45.2	<code>basic_raw_socket::operator=</code> (2 of 2 overloads)	412
5.71.46	<code>basic_raw_socket::out_of_band_inline</code>	412
5.71.47	<code>basic_raw_socket::protocol_type</code>	413
5.71.48	<code>basic_raw_socket::receive</code>	413
5.71.48.1	<code>basic_raw_socket::receive</code> (1 of 3 overloads)	414
5.71.48.2	<code>basic_raw_socket::receive</code> (2 of 3 overloads)	414
5.71.48.3	<code>basic_raw_socket::receive</code> (3 of 3 overloads)	415
5.71.49	<code>basic_raw_socket::receive_buffer_size</code>	415
5.71.50	<code>basic_raw_socket::receive_from</code>	416
5.71.50.1	<code>basic_raw_socket::receive_from</code> (1 of 3 overloads)	416
5.71.50.2	<code>basic_raw_socket::receive_from</code> (2 of 3 overloads)	417
5.71.50.3	<code>basic_raw_socket::receive_from</code> (3 of 3 overloads)	418
5.71.51	<code>basic_raw_socket::receive_low_watermark</code>	418
5.71.52	<code>basic_raw_socket::release</code>	419
5.71.52.1	<code>basic_raw_socket::release</code> (1 of 2 overloads)	419

5.71.52.2	<code>basic_raw_socket::release</code> (2 of 2 overloads)	419
5.71.53	<code>basic_raw_socket::remote_endpoint</code>	420
5.71.53.1	<code>basic_raw_socket::remote_endpoint</code> (1 of 2 overloads)	420
5.71.53.2	<code>basic_raw_socket::remote_endpoint</code> (2 of 2 overloads)	420
5.71.54	<code>basic_raw_socket::reuse_address</code>	421
5.71.55	<code>basic_raw_socket::send</code>	421
5.71.55.1	<code>basic_raw_socket::send</code> (1 of 3 overloads)	422
5.71.55.2	<code>basic_raw_socket::send</code> (2 of 3 overloads)	423
5.71.55.3	<code>basic_raw_socket::send</code> (3 of 3 overloads)	423
5.71.56	<code>basic_raw_socket::send_buffer_size</code>	424
5.71.57	<code>basic_raw_socket::send_low_watermark</code>	424
5.71.58	<code>basic_raw_socket::send_to</code>	425
5.71.58.1	<code>basic_raw_socket::send_to</code> (1 of 3 overloads)	425
5.71.58.2	<code>basic_raw_socket::send_to</code> (2 of 3 overloads)	426
5.71.58.3	<code>basic_raw_socket::send_to</code> (3 of 3 overloads)	427
5.71.59	<code>basic_raw_socket::set_option</code>	427
5.71.59.1	<code>basic_raw_socket::set_option</code> (1 of 2 overloads)	427
5.71.59.2	<code>basic_raw_socket::set_option</code> (2 of 2 overloads)	428
5.71.60	<code>basic_raw_socket::shutdown</code>	429
5.71.60.1	<code>basic_raw_socket::shutdown</code> (1 of 2 overloads)	429
5.71.60.2	<code>basic_raw_socket::shutdown</code> (2 of 2 overloads)	429
5.71.61	<code>basic_raw_socket::shutdown_type</code>	430
5.71.62	<code>basic_raw_socket::wait</code>	430
5.71.62.1	<code>basic_raw_socket::wait</code> (1 of 2 overloads)	430
5.71.62.2	<code>basic_raw_socket::wait</code> (2 of 2 overloads)	431
5.71.63	<code>basic_raw_socket::wait_type</code>	431
5.71.64	<code>basic_raw_socket::~basic_raw_socket</code>	432
5.72	<code>basic_seq_packet_socket</code>	432
5.72.1	<code>basic_seq_packet_socket::assign</code>	435
5.72.1.1	<code>basic_seq_packet_socket::assign</code> (1 of 2 overloads)	435
5.72.1.2	<code>basic_seq_packet_socket::assign</code> (2 of 2 overloads)	436
5.72.2	<code>basic_seq_packet_socket::async_connect</code>	436
5.72.3	<code>basic_seq_packet_socket::async_receive</code>	437
5.72.3.1	<code>basic_seq_packet_socket::async_receive</code> (1 of 2 overloads)	437
5.72.3.2	<code>basic_seq_packet_socket::async_receive</code> (2 of 2 overloads)	438
5.72.4	<code>basic_seq_packet_socket::async_send</code>	439
5.72.5	<code>basic_seq_packet_socket::async_wait</code>	439
5.72.6	<code>basic_seq_packet_socket::at_mark</code>	440
5.72.6.1	<code>basic_seq_packet_socket::at_mark</code> (1 of 2 overloads)	440

5.72.6.2	<code>basic_seq_packet_socket::at_mark</code> (2 of 2 overloads)	441
5.72.7	<code>basic_seq_packet_socket::available</code>	441
5.72.7.1	<code>basic_seq_packet_socket::available</code> (1 of 2 overloads)	441
5.72.7.2	<code>basic_seq_packet_socket::available</code> (2 of 2 overloads)	442
5.72.8	<code>basic_seq_packet_socket::basic_seq_packet_socket</code>	442
5.72.8.1	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (1 of 6 overloads)	443
5.72.8.2	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (2 of 6 overloads)	443
5.72.8.3	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (3 of 6 overloads)	443
5.72.8.4	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (4 of 6 overloads)	444
5.72.8.5	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (5 of 6 overloads)	444
5.72.8.6	<code>basic_seq_packet_socket::basic_seq_packet_socket</code> (6 of 6 overloads)	444
5.72.9	<code>basic_seq_packet_socket::bind</code>	445
5.72.9.1	<code>basic_seq_packet_socket::bind</code> (1 of 2 overloads)	445
5.72.9.2	<code>basic_seq_packet_socket::bind</code> (2 of 2 overloads)	446
5.72.10	<code>basic_seq_packet_socket::broadcast</code>	446
5.72.11	<code>basic_seq_packet_socket::bytes_readable</code>	447
5.72.12	<code>basic_seq_packet_socket::cancel</code>	447
5.72.12.1	<code>basic_seq_packet_socket::cancel</code> (1 of 2 overloads)	447
5.72.12.2	<code>basic_seq_packet_socket::cancel</code> (2 of 2 overloads)	448
5.72.13	<code>basic_seq_packet_socket::close</code>	449
5.72.13.1	<code>basic_seq_packet_socket::close</code> (1 of 2 overloads)	449
5.72.13.2	<code>basic_seq_packet_socket::close</code> (2 of 2 overloads)	449
5.72.14	<code>basic_seq_packet_socket::connect</code>	450
5.72.14.1	<code>basic_seq_packet_socket::connect</code> (1 of 2 overloads)	450
5.72.14.2	<code>basic_seq_packet_socket::connect</code> (2 of 2 overloads)	451
5.72.15	<code>basic_seq_packet_socket::debug</code>	451
5.72.16	<code>basic_seq_packet_socket::do_not_route</code>	452
5.72.17	<code>basic_seq_packet_socket::enable_connection_aborted</code>	452
5.72.18	<code>basic_seq_packet_socket::endpoint_type</code>	453
5.72.19	<code>basic_seq_packet_socket::executor_type</code>	453
5.72.20	<code>basic_seq_packet_socket::get_executor</code>	454
5.72.21	<code>basic_seq_packet_socket::get_io_context</code>	454
5.72.22	<code>basic_seq_packet_socket::get_io_service</code>	454
5.72.23	<code>basic_seq_packet_socket::get_option</code>	455
5.72.23.1	<code>basic_seq_packet_socket::get_option</code> (1 of 2 overloads)	455
5.72.23.2	<code>basic_seq_packet_socket::get_option</code> (2 of 2 overloads)	456
5.72.24	<code>basic_seq_packet_socket::io_control</code>	456
5.72.24.1	<code>basic_seq_packet_socket::io_control</code> (1 of 2 overloads)	457
5.72.24.2	<code>basic_seq_packet_socket::io_control</code> (2 of 2 overloads)	457

5.72.25	basic_seq_packet_socket::is_open	458
5.72.26	basic_seq_packet_socket::keep_alive	458
5.72.27	basic_seq_packet_socket::linger	459
5.72.28	basic_seq_packet_socket::local_endpoint	459
5.72.28.1	basic_seq_packet_socket::local_endpoint (1 of 2 overloads)	459
5.72.28.2	basic_seq_packet_socket::local_endpoint (2 of 2 overloads)	460
5.72.29	basic_seq_packet_socket::lowest_layer	460
5.72.29.1	basic_seq_packet_socket::lowest_layer (1 of 2 overloads)	461
5.72.29.2	basic_seq_packet_socket::lowest_layer (2 of 2 overloads)	461
5.72.30	basic_seq_packet_socket::lowest_layer_type	461
5.72.31	basic_seq_packet_socket::max_connections	465
5.72.32	basic_seq_packet_socket::max_listen_connections	465
5.72.33	basic_seq_packet_socket::message_do_not_route	465
5.72.34	basic_seq_packet_socket::message_end_of_record	465
5.72.35	basic_seq_packet_socket::message_flags	465
5.72.36	basic_seq_packet_socket::message_out_of_band	465
5.72.37	basic_seq_packet_socket::message_peek	466
5.72.38	basic_seq_packet_socket::native_handle	466
5.72.39	basic_seq_packet_socket::native_handle_type	466
5.72.40	basic_seq_packet_socket::native_non_blocking	466
5.72.40.1	basic_seq_packet_socket::native_non_blocking (1 of 3 overloads)	466
5.72.40.2	basic_seq_packet_socket::native_non_blocking (2 of 3 overloads)	468
5.72.40.3	basic_seq_packet_socket::native_non_blocking (3 of 3 overloads)	469
5.72.41	basic_seq_packet_socket::non_blocking	471
5.72.41.1	basic_seq_packet_socket::non_blocking (1 of 3 overloads)	471
5.72.41.2	basic_seq_packet_socket::non_blocking (2 of 3 overloads)	472
5.72.41.3	basic_seq_packet_socket::non_blocking (3 of 3 overloads)	472
5.72.42	basic_seq_packet_socket::open	473
5.72.42.1	basic_seq_packet_socket::open (1 of 2 overloads)	473
5.72.42.2	basic_seq_packet_socket::open (2 of 2 overloads)	473
5.72.43	basic_seq_packet_socket::operator=	474
5.72.43.1	basic_seq_packet_socket::operator= (1 of 2 overloads)	474
5.72.43.2	basic_seq_packet_socket::operator= (2 of 2 overloads)	474
5.72.44	basic_seq_packet_socket::out_of_band_inline	475
5.72.45	basic_seq_packet_socket::protocol_type	475
5.72.46	basic_seq_packet_socket::receive	476
5.72.46.1	basic_seq_packet_socket::receive (1 of 3 overloads)	476
5.72.46.2	basic_seq_packet_socket::receive (2 of 3 overloads)	477
5.72.46.3	basic_seq_packet_socket::receive (3 of 3 overloads)	478

5.72.47	<code>basic_seq_packet_socket::receive_buffer_size</code>	478
5.72.48	<code>basic_seq_packet_socket::receive_low_watermark</code>	479
5.72.49	<code>basic_seq_packet_socket::release</code>	479
5.72.49.1	<code>basic_seq_packet_socket::release</code> (1 of 2 overloads)	479
5.72.49.2	<code>basic_seq_packet_socket::release</code> (2 of 2 overloads)	480
5.72.50	<code>basic_seq_packet_socket::remote_endpoint</code>	480
5.72.50.1	<code>basic_seq_packet_socket::remote_endpoint</code> (1 of 2 overloads)	480
5.72.50.2	<code>basic_seq_packet_socket::remote_endpoint</code> (2 of 2 overloads)	481
5.72.51	<code>basic_seq_packet_socket::reuse_address</code>	481
5.72.52	<code>basic_seq_packet_socket::send</code>	482
5.72.52.1	<code>basic_seq_packet_socket::send</code> (1 of 2 overloads)	482
5.72.52.2	<code>basic_seq_packet_socket::send</code> (2 of 2 overloads)	483
5.72.53	<code>basic_seq_packet_socket::send_buffer_size</code>	483
5.72.54	<code>basic_seq_packet_socket::send_low_watermark</code>	484
5.72.55	<code>basic_seq_packet_socket::set_option</code>	485
5.72.55.1	<code>basic_seq_packet_socket::set_option</code> (1 of 2 overloads)	485
5.72.55.2	<code>basic_seq_packet_socket::set_option</code> (2 of 2 overloads)	485
5.72.56	<code>basic_seq_packet_socket::shutdown</code>	486
5.72.56.1	<code>basic_seq_packet_socket::shutdown</code> (1 of 2 overloads)	486
5.72.56.2	<code>basic_seq_packet_socket::shutdown</code> (2 of 2 overloads)	487
5.72.57	<code>basic_seq_packet_socket::shutdown_type</code>	487
5.72.58	<code>basic_seq_packet_socket::wait</code>	488
5.72.58.1	<code>basic_seq_packet_socket::wait</code> (1 of 2 overloads)	488
5.72.58.2	<code>basic_seq_packet_socket::wait</code> (2 of 2 overloads)	488
5.72.59	<code>basic_seq_packet_socket::wait_type</code>	489
5.72.60	<code>basic_seq_packet_socket::~basic_seq_packet_socket</code>	489
5.73	<code>basic_socket</code>	489
5.73.1	<code>basic_socket::assign</code>	493
5.73.1.1	<code>basic_socket::assign</code> (1 of 2 overloads)	493
5.73.1.2	<code>basic_socket::assign</code> (2 of 2 overloads)	493
5.73.2	<code>basic_socket::async_connect</code>	493
5.73.3	<code>basic_socket::async_wait</code>	494
5.73.4	<code>basic_socket::at_mark</code>	495
5.73.4.1	<code>basic_socket::at_mark</code> (1 of 2 overloads)	495
5.73.4.2	<code>basic_socket::at_mark</code> (2 of 2 overloads)	495
5.73.5	<code>basic_socket::available</code>	496
5.73.5.1	<code>basic_socket::available</code> (1 of 2 overloads)	496
5.73.5.2	<code>basic_socket::available</code> (2 of 2 overloads)	496
5.73.6	<code>basic_socket::basic_socket</code>	497

5.73.6.1	basic_socket::basic_socket (1 of 6 overloads)	497
5.73.6.2	basic_socket::basic_socket (2 of 6 overloads)	498
5.73.6.3	basic_socket::basic_socket (3 of 6 overloads)	498
5.73.6.4	basic_socket::basic_socket (4 of 6 overloads)	498
5.73.6.5	basic_socket::basic_socket (5 of 6 overloads)	499
5.73.6.6	basic_socket::basic_socket (6 of 6 overloads)	499
5.73.7	basic_socket::bind	500
5.73.7.1	basic_socket::bind (1 of 2 overloads)	500
5.73.7.2	basic_socket::bind (2 of 2 overloads)	500
5.73.8	basic_socket::broadcast	501
5.73.9	basic_socket::bytes_readable	501
5.73.10	basic_socket::cancel	502
5.73.10.1	basic_socket::cancel (1 of 2 overloads)	502
5.73.10.2	basic_socket::cancel (2 of 2 overloads)	503
5.73.11	basic_socket::close	503
5.73.11.1	basic_socket::close (1 of 2 overloads)	503
5.73.11.2	basic_socket::close (2 of 2 overloads)	504
5.73.12	basic_socket::connect	504
5.73.12.1	basic_socket::connect (1 of 2 overloads)	505
5.73.12.2	basic_socket::connect (2 of 2 overloads)	505
5.73.13	basic_socket::debug	506
5.73.14	basic_socket::do_not_route	506
5.73.15	basic_socket::enable_connection_aborted	507
5.73.16	basic_socket::endpoint_type	507
5.73.17	basic_socket::executor_type	507
5.73.18	basic_socket::get_executor	508
5.73.19	basic_socket::get_io_context	508
5.73.20	basic_socket::get_io_service	509
5.73.21	basic_socket::get_option	509
5.73.21.1	basic_socket::get_option (1 of 2 overloads)	509
5.73.21.2	basic_socket::get_option (2 of 2 overloads)	510
5.73.22	basic_socket::io_control	510
5.73.22.1	basic_socket::io_control (1 of 2 overloads)	510
5.73.22.2	basic_socket::io_control (2 of 2 overloads)	511
5.73.23	basic_socket::is_open	511
5.73.24	basic_socket::keep_alive	512
5.73.25	basic_socket::linger	512
5.73.26	basic_socket::local_endpoint	513
5.73.26.1	basic_socket::local_endpoint (1 of 2 overloads)	513

5.73.26.2	<code>basic_socket::local_endpoint</code> (2 of 2 overloads)	513
5.73.27	<code>basic_socket::lowest_layer</code>	514
5.73.27.1	<code>basic_socket::lowest_layer</code> (1 of 2 overloads)	514
5.73.27.2	<code>basic_socket::lowest_layer</code> (2 of 2 overloads)	514
5.73.28	<code>basic_socket::lowest_layer_type</code>	515
5.73.29	<code>basic_socket::max_connections</code>	518
5.73.30	<code>basic_socket::max_listen_connections</code>	518
5.73.31	<code>basic_socket::message_do_not_route</code>	518
5.73.32	<code>basic_socket::message_end_of_record</code>	518
5.73.33	<code>basic_socket::message_flags</code>	518
5.73.34	<code>basic_socket::message_out_of_band</code>	519
5.73.35	<code>basic_socket::message_peek</code>	519
5.73.36	<code>basic_socket::native_handle</code>	519
5.73.37	<code>basic_socket::native_handle_type</code>	519
5.73.38	<code>basic_socket::native_non_blocking</code>	519
5.73.38.1	<code>basic_socket::native_non_blocking</code> (1 of 3 overloads)	520
5.73.38.2	<code>basic_socket::native_non_blocking</code> (2 of 3 overloads)	521
5.73.38.3	<code>basic_socket::native_non_blocking</code> (3 of 3 overloads)	523
5.73.39	<code>basic_socket::non_blocking</code>	524
5.73.39.1	<code>basic_socket::non_blocking</code> (1 of 3 overloads)	524
5.73.39.2	<code>basic_socket::non_blocking</code> (2 of 3 overloads)	525
5.73.39.3	<code>basic_socket::non_blocking</code> (3 of 3 overloads)	525
5.73.40	<code>basic_socket::open</code>	526
5.73.40.1	<code>basic_socket::open</code> (1 of 2 overloads)	526
5.73.40.2	<code>basic_socket::open</code> (2 of 2 overloads)	526
5.73.41	<code>basic_socket::operator=</code>	527
5.73.41.1	<code>basic_socket::operator=</code> (1 of 2 overloads)	527
5.73.41.2	<code>basic_socket::operator=</code> (2 of 2 overloads)	527
5.73.42	<code>basic_socket::out_of_band_inline</code>	528
5.73.43	<code>basic_socket::protocol_type</code>	528
5.73.44	<code>basic_socket::receive_buffer_size</code>	529
5.73.45	<code>basic_socket::receive_low_watermark</code>	529
5.73.46	<code>basic_socket::release</code>	530
5.73.46.1	<code>basic_socket::release</code> (1 of 2 overloads)	530
5.73.46.2	<code>basic_socket::release</code> (2 of 2 overloads)	530
5.73.47	<code>basic_socket::remote_endpoint</code>	531
5.73.47.1	<code>basic_socket::remote_endpoint</code> (1 of 2 overloads)	531
5.73.47.2	<code>basic_socket::remote_endpoint</code> (2 of 2 overloads)	531
5.73.48	<code>basic_socket::reuse_address</code>	532

5.73.49	<code>basic_socket::send_buffer_size</code>	532
5.73.50	<code>basic_socket::send_low_watermark</code>	533
5.73.51	<code>basic_socket::set_option</code>	534
5.73.51.1	<code>basic_socket::set_option</code> (1 of 2 overloads)	534
5.73.51.2	<code>basic_socket::set_option</code> (2 of 2 overloads)	534
5.73.52	<code>basic_socket::shutdown</code>	535
5.73.52.1	<code>basic_socket::shutdown</code> (1 of 2 overloads)	535
5.73.52.2	<code>basic_socket::shutdown</code> (2 of 2 overloads)	536
5.73.53	<code>basic_socket::shutdown_type</code>	536
5.73.54	<code>basic_socket::wait</code>	536
5.73.54.1	<code>basic_socket::wait</code> (1 of 2 overloads)	537
5.73.54.2	<code>basic_socket::wait</code> (2 of 2 overloads)	537
5.73.55	<code>basic_socket::wait_type</code>	537
5.73.56	<code>basic_socket::~basic_socket</code>	538
5.74	<code>basic_socket_acceptor</code>	538
5.74.1	<code>basic_socket_acceptor::accept</code>	541
5.74.1.1	<code>basic_socket_acceptor::accept</code> (1 of 12 overloads)	542
5.74.1.2	<code>basic_socket_acceptor::accept</code> (2 of 12 overloads)	543
5.74.1.3	<code>basic_socket_acceptor::accept</code> (3 of 12 overloads)	543
5.74.1.4	<code>basic_socket_acceptor::accept</code> (4 of 12 overloads)	544
5.74.1.5	<code>basic_socket_acceptor::accept</code> (5 of 12 overloads)	545
5.74.1.6	<code>basic_socket_acceptor::accept</code> (6 of 12 overloads)	545
5.74.1.7	<code>basic_socket_acceptor::accept</code> (7 of 12 overloads)	546
5.74.1.8	<code>basic_socket_acceptor::accept</code> (8 of 12 overloads)	546
5.74.1.9	<code>basic_socket_acceptor::accept</code> (9 of 12 overloads)	547
5.74.1.10	<code>basic_socket_acceptor::accept</code> (10 of 12 overloads)	547
5.74.1.11	<code>basic_socket_acceptor::accept</code> (11 of 12 overloads)	548
5.74.1.12	<code>basic_socket_acceptor::accept</code> (12 of 12 overloads)	549
5.74.2	<code>basic_socket_acceptor::assign</code>	549
5.74.2.1	<code>basic_socket_acceptor::assign</code> (1 of 2 overloads)	550
5.74.2.2	<code>basic_socket_acceptor::assign</code> (2 of 2 overloads)	550
5.74.3	<code>basic_socket_acceptor::async_accept</code>	550
5.74.3.1	<code>basic_socket_acceptor::async_accept</code> (1 of 6 overloads)	551
5.74.3.2	<code>basic_socket_acceptor::async_accept</code> (2 of 6 overloads)	552
5.74.3.3	<code>basic_socket_acceptor::async_accept</code> (3 of 6 overloads)	552
5.74.3.4	<code>basic_socket_acceptor::async_accept</code> (4 of 6 overloads)	553
5.74.3.5	<code>basic_socket_acceptor::async_accept</code> (5 of 6 overloads)	554
5.74.3.6	<code>basic_socket_acceptor::async_accept</code> (6 of 6 overloads)	555
5.74.4	<code>basic_socket_acceptor::async_wait</code>	555

5.74.5	<code>basic_socket_acceptor::basic_socket_acceptor</code>	556
5.74.5.1	<code>basic_socket_acceptor::basic_socket_acceptor</code> (1 of 6 overloads)	557
5.74.5.2	<code>basic_socket_acceptor::basic_socket_acceptor</code> (2 of 6 overloads)	557
5.74.5.3	<code>basic_socket_acceptor::basic_socket_acceptor</code> (3 of 6 overloads)	558
5.74.5.4	<code>basic_socket_acceptor::basic_socket_acceptor</code> (4 of 6 overloads)	558
5.74.5.5	<code>basic_socket_acceptor::basic_socket_acceptor</code> (5 of 6 overloads)	559
5.74.5.6	<code>basic_socket_acceptor::basic_socket_acceptor</code> (6 of 6 overloads)	559
5.74.6	<code>basic_socket_acceptor::bind</code>	559
5.74.6.1	<code>basic_socket_acceptor::bind</code> (1 of 2 overloads)	560
5.74.6.2	<code>basic_socket_acceptor::bind</code> (2 of 2 overloads)	560
5.74.7	<code>basic_socket_acceptor::broadcast</code>	561
5.74.8	<code>basic_socket_acceptor::bytes_readable</code>	561
5.74.9	<code>basic_socket_acceptor::cancel</code>	562
5.74.9.1	<code>basic_socket_acceptor::cancel</code> (1 of 2 overloads)	562
5.74.9.2	<code>basic_socket_acceptor::cancel</code> (2 of 2 overloads)	562
5.74.10	<code>basic_socket_acceptor::close</code>	562
5.74.10.1	<code>basic_socket_acceptor::close</code> (1 of 2 overloads)	562
5.74.10.2	<code>basic_socket_acceptor::close</code> (2 of 2 overloads)	563
5.74.11	<code>basic_socket_acceptor::debug</code>	563
5.74.12	<code>basic_socket_acceptor::do_not_route</code>	564
5.74.13	<code>basic_socket_acceptor::enable_connection_aborted</code>	564
5.74.14	<code>basic_socket_acceptor::endpoint_type</code>	565
5.74.15	<code>basic_socket_acceptor::executor_type</code>	565
5.74.16	<code>basic_socket_acceptor::get_executor</code>	566
5.74.17	<code>basic_socket_acceptor::get_io_context</code>	566
5.74.18	<code>basic_socket_acceptor::get_io_service</code>	566
5.74.19	<code>basic_socket_acceptor::get_option</code>	567
5.74.19.1	<code>basic_socket_acceptor::get_option</code> (1 of 2 overloads)	567
5.74.19.2	<code>basic_socket_acceptor::get_option</code> (2 of 2 overloads)	568
5.74.20	<code>basic_socket_acceptor::io_control</code>	568
5.74.20.1	<code>basic_socket_acceptor::io_control</code> (1 of 2 overloads)	568
5.74.20.2	<code>basic_socket_acceptor::io_control</code> (2 of 2 overloads)	569
5.74.21	<code>basic_socket_acceptor::is_open</code>	569
5.74.22	<code>basic_socket_acceptor::keep_alive</code>	570
5.74.23	<code>basic_socket_acceptor::linger</code>	570
5.74.24	<code>basic_socket_acceptor::listen</code>	571
5.74.24.1	<code>basic_socket_acceptor::listen</code> (1 of 2 overloads)	571
5.74.24.2	<code>basic_socket_acceptor::listen</code> (2 of 2 overloads)	571
5.74.25	<code>basic_socket_acceptor::local_endpoint</code>	572

5.74.25.1	basic_socket_acceptor::local_endpoint (1 of 2 overloads)	572
5.74.25.2	basic_socket_acceptor::local_endpoint (2 of 2 overloads)	572
5.74.26	basic_socket_acceptor::max_connections	573
5.74.27	basic_socket_acceptor::max_listen_connections	573
5.74.28	basic_socket_acceptor::message_do_not_route	573
5.74.29	basic_socket_acceptor::message_end_of_record	573
5.74.30	basic_socket_acceptor::message_flags	574
5.74.31	basic_socket_acceptor::message_out_of_band	574
5.74.32	basic_socket_acceptor::message_peek	574
5.74.33	basic_socket_acceptor::native_handle	574
5.74.34	basic_socket_acceptor::native_handle_type	574
5.74.35	basic_socket_acceptor::native_non_blocking	575
5.74.35.1	basic_socket_acceptor::native_non_blocking (1 of 3 overloads)	575
5.74.35.2	basic_socket_acceptor::native_non_blocking (2 of 3 overloads)	575
5.74.35.3	basic_socket_acceptor::native_non_blocking (3 of 3 overloads)	576
5.74.36	basic_socket_acceptor::non_blocking	576
5.74.36.1	basic_socket_acceptor::non_blocking (1 of 3 overloads)	576
5.74.36.2	basic_socket_acceptor::non_blocking (2 of 3 overloads)	577
5.74.36.3	basic_socket_acceptor::non_blocking (3 of 3 overloads)	577
5.74.37	basic_socket_acceptor::open	577
5.74.37.1	basic_socket_acceptor::open (1 of 2 overloads)	578
5.74.37.2	basic_socket_acceptor::open (2 of 2 overloads)	578
5.74.38	basic_socket_acceptor::operator=	579
5.74.38.1	basic_socket_acceptor::operator= (1 of 2 overloads)	579
5.74.38.2	basic_socket_acceptor::operator= (2 of 2 overloads)	579
5.74.39	basic_socket_acceptor::out_of_band_inline	580
5.74.40	basic_socket_acceptor::protocol_type	580
5.74.41	basic_socket_acceptor::receive_buffer_size	580
5.74.42	basic_socket_acceptor::receive_low_watermark	581
5.74.43	basic_socket_acceptor::release	582
5.74.43.1	basic_socket_acceptor::release (1 of 2 overloads)	582
5.74.43.2	basic_socket_acceptor::release (2 of 2 overloads)	582
5.74.44	basic_socket_acceptor::reuse_address	583
5.74.45	basic_socket_acceptor::send_buffer_size	583
5.74.46	basic_socket_acceptor::send_low_watermark	584
5.74.47	basic_socket_acceptor::set_option	584
5.74.47.1	basic_socket_acceptor::set_option (1 of 2 overloads)	585
5.74.47.2	basic_socket_acceptor::set_option (2 of 2 overloads)	585
5.74.48	basic_socket_acceptor::shutdown_type	586

5.74.49 basic_socket_acceptor::wait	586
5.74.49.1 basic_socket_acceptor::wait (1 of 2 overloads)	586
5.74.49.2 basic_socket_acceptor::wait (2 of 2 overloads)	586
5.74.50 basic_socket_acceptor::wait_type	587
5.74.51 basic_socket_acceptor::~basic_socket_acceptor	587
5.75 basic_socket_iostream	587
5.75.1 basic_socket_iostream::basic_socket_iostream	589
5.75.1.1 basic_socket_iostream::basic_socket_iostream (1 of 4 overloads)	589
5.75.1.2 basic_socket_iostream::basic_socket_iostream (2 of 4 overloads)	589
5.75.1.3 basic_socket_iostream::basic_socket_iostream (3 of 4 overloads)	589
5.75.1.4 basic_socket_iostream::basic_socket_iostream (4 of 4 overloads)	590
5.75.2 basic_socket_iostream::clock_type	590
5.75.3 basic_socket_iostream::close	590
5.75.4 basic_socket_iostream::connect	590
5.75.5 basic_socket_iostream::duration	590
5.75.6 basic_socket_iostream::duration_type	591
5.75.7 basic_socket_iostream::endpoint_type	591
5.75.8 basic_socket_iostream::error	591
5.75.9 basic_socket_iostream::expires_after	592
5.75.10 basic_socket_iostream::expires_at	592
5.75.10.1 basic_socket_iostream::expires_at (1 of 2 overloads)	592
5.75.10.2 basic_socket_iostream::expires_at (2 of 2 overloads)	592
5.75.11 basic_socket_iostream::expires_from_now	593
5.75.11.1 basic_socket_iostream::expires_from_now (1 of 2 overloads)	593
5.75.11.2 basic_socket_iostream::expires_from_now (2 of 2 overloads)	593
5.75.12 basic_socket_iostream::expiry	593
5.75.13 basic_socket_iostream::operator=	593
5.75.14 basic_socket_iostream::protocol_type	594
5.75.15 basic_socket_iostream::rdbuf	594
5.75.16 basic_socket_iostream::socket	594
5.75.17 basic_socket_iostream::time_point	594
5.75.18 basic_socket_iostream::time_type	594
5.76 basic_socket_streambuf	595
5.76.1 basic_socket_streambuf::basic_socket_streambuf	596
5.76.1.1 basic_socket_streambuf::basic_socket_streambuf (1 of 3 overloads)	597
5.76.1.2 basic_socket_streambuf::basic_socket_streambuf (2 of 3 overloads)	597
5.76.1.3 basic_socket_streambuf::basic_socket_streambuf (3 of 3 overloads)	597
5.76.2 basic_socket_streambuf::clock_type	597
5.76.3 basic_socket_streambuf::close	597

5.76.4	<code>basic_socket_streambuf::connect</code>	597
5.76.4.1	<code>basic_socket_streambuf::connect</code> (1 of 2 overloads)	598
5.76.4.2	<code>basic_socket_streambuf::connect</code> (2 of 2 overloads)	598
5.76.5	<code>basic_socket_streambuf::duration</code>	598
5.76.6	<code>basic_socket_streambuf::duration_type</code>	598
5.76.7	<code>basic_socket_streambuf::endpoint_type</code>	599
5.76.8	<code>basic_socket_streambuf::error</code>	599
5.76.9	<code>basic_socket_streambuf::expires_after</code>	599
5.76.10	<code>basic_socket_streambuf::expires_at</code>	599
5.76.10.1	<code>basic_socket_streambuf::expires_at</code> (1 of 2 overloads)	599
5.76.10.2	<code>basic_socket_streambuf::expires_at</code> (2 of 2 overloads)	600
5.76.11	<code>basic_socket_streambuf::expires_from_now</code>	600
5.76.11.1	<code>basic_socket_streambuf::expires_from_now</code> (1 of 2 overloads)	600
5.76.11.2	<code>basic_socket_streambuf::expires_from_now</code> (2 of 2 overloads)	600
5.76.12	<code>basic_socket_streambuf::expiry</code>	601
5.76.13	<code>basic_socket_streambuf::operator=</code>	601
5.76.14	<code>basic_socket_streambuf::overflow</code>	601
5.76.15	<code>basic_socket_streambuf::protocol_type</code>	601
5.76.16	<code>basic_socket_streambuf::puberror</code>	601
5.76.17	<code>basic_socket_streambuf::setbuf</code>	601
5.76.18	<code>basic_socket_streambuf::socket</code>	602
5.76.19	<code>basic_socket_streambuf::sync</code>	602
5.76.20	<code>basic_socket_streambuf::time_point</code>	602
5.76.21	<code>basic_socket_streambuf::time_type</code>	602
5.76.22	<code>basic_socket_streambuf::underflow</code>	602
5.76.23	<code>basic_socket_streambuf::~basic_socket_streambuf</code>	602
5.77	<code>basic_stream_socket</code>	602
5.77.1	<code>basic_stream_socket::assign</code>	606
5.77.1.1	<code>basic_stream_socket::assign</code> (1 of 2 overloads)	606
5.77.1.2	<code>basic_stream_socket::assign</code> (2 of 2 overloads)	606
5.77.2	<code>basic_stream_socket::async_connect</code>	607
5.77.3	<code>basic_stream_socket::async_read_some</code>	607
5.77.4	<code>basic_stream_socket::async_receive</code>	608
5.77.4.1	<code>basic_stream_socket::async_receive</code> (1 of 2 overloads)	609
5.77.4.2	<code>basic_stream_socket::async_receive</code> (2 of 2 overloads)	609
5.77.5	<code>basic_stream_socket::async_send</code>	610
5.77.5.1	<code>basic_stream_socket::async_send</code> (1 of 2 overloads)	611
5.77.5.2	<code>basic_stream_socket::async_send</code> (2 of 2 overloads)	611
5.77.6	<code>basic_stream_socket::async_wait</code>	612

5.77.7	<code>basic_stream_socket::async_write_some</code>	613
5.77.8	<code>basic_stream_socket::at_mark</code>	614
5.77.8.1	<code>basic_stream_socket::at_mark</code> (1 of 2 overloads)	614
5.77.8.2	<code>basic_stream_socket::at_mark</code> (2 of 2 overloads)	614
5.77.9	<code>basic_stream_socket::available</code>	614
5.77.9.1	<code>basic_stream_socket::available</code> (1 of 2 overloads)	615
5.77.9.2	<code>basic_stream_socket::available</code> (2 of 2 overloads)	615
5.77.10	<code>basic_stream_socket::basic_stream_socket</code>	615
5.77.10.1	<code>basic_stream_socket::basic_stream_socket</code> (1 of 6 overloads)	616
5.77.10.2	<code>basic_stream_socket::basic_stream_socket</code> (2 of 6 overloads)	616
5.77.10.3	<code>basic_stream_socket::basic_stream_socket</code> (3 of 6 overloads)	617
5.77.10.4	<code>basic_stream_socket::basic_stream_socket</code> (4 of 6 overloads)	617
5.77.10.5	<code>basic_stream_socket::basic_stream_socket</code> (5 of 6 overloads)	618
5.77.10.6	<code>basic_stream_socket::basic_stream_socket</code> (6 of 6 overloads)	618
5.77.11	<code>basic_stream_socket::bind</code>	618
5.77.11.1	<code>basic_stream_socket::bind</code> (1 of 2 overloads)	619
5.77.11.2	<code>basic_stream_socket::bind</code> (2 of 2 overloads)	619
5.77.12	<code>basic_stream_socket::broadcast</code>	620
5.77.13	<code>basic_stream_socket::bytes_readable</code>	620
5.77.14	<code>basic_stream_socket::cancel</code>	621
5.77.14.1	<code>basic_stream_socket::cancel</code> (1 of 2 overloads)	621
5.77.14.2	<code>basic_stream_socket::cancel</code> (2 of 2 overloads)	621
5.77.15	<code>basic_stream_socket::close</code>	622
5.77.15.1	<code>basic_stream_socket::close</code> (1 of 2 overloads)	622
5.77.15.2	<code>basic_stream_socket::close</code> (2 of 2 overloads)	623
5.77.16	<code>basic_stream_socket::connect</code>	623
5.77.16.1	<code>basic_stream_socket::connect</code> (1 of 2 overloads)	623
5.77.16.2	<code>basic_stream_socket::connect</code> (2 of 2 overloads)	624
5.77.17	<code>basic_stream_socket::debug</code>	624
5.77.18	<code>basic_stream_socket::do_not_route</code>	625
5.77.19	<code>basic_stream_socket::enable_connection_aborted</code>	626
5.77.20	<code>basic_stream_socket::endpoint_type</code>	626
5.77.21	<code>basic_stream_socket::executor_type</code>	626
5.77.22	<code>basic_stream_socket::get_executor</code>	627
5.77.23	<code>basic_stream_socket::get_io_context</code>	627
5.77.24	<code>basic_stream_socket::get_io_service</code>	628
5.77.25	<code>basic_stream_socket::get_option</code>	628
5.77.25.1	<code>basic_stream_socket::get_option</code> (1 of 2 overloads)	628
5.77.25.2	<code>basic_stream_socket::get_option</code> (2 of 2 overloads)	629

5.77.26	<code>basic_stream_socket::io_control</code>	629
5.77.26.1	<code>basic_stream_socket::io_control</code> (1 of 2 overloads)	630
5.77.26.2	<code>basic_stream_socket::io_control</code> (2 of 2 overloads)	630
5.77.27	<code>basic_stream_socket::is_open</code>	631
5.77.28	<code>basic_stream_socket::keep_alive</code>	631
5.77.29	<code>basic_stream_socket::linger</code>	632
5.77.30	<code>basic_stream_socket::local_endpoint</code>	632
5.77.30.1	<code>basic_stream_socket::local_endpoint</code> (1 of 2 overloads)	632
5.77.30.2	<code>basic_stream_socket::local_endpoint</code> (2 of 2 overloads)	633
5.77.31	<code>basic_stream_socket::lowest_layer</code>	633
5.77.31.1	<code>basic_stream_socket::lowest_layer</code> (1 of 2 overloads)	634
5.77.31.2	<code>basic_stream_socket::lowest_layer</code> (2 of 2 overloads)	634
5.77.32	<code>basic_stream_socket::lowest_layer_type</code>	634
5.77.33	<code>basic_stream_socket::max_connections</code>	638
5.77.34	<code>basic_stream_socket::max_listen_connections</code>	638
5.77.35	<code>basic_stream_socket::message_do_not_route</code>	638
5.77.36	<code>basic_stream_socket::message_end_of_record</code>	638
5.77.37	<code>basic_stream_socket::message_flags</code>	638
5.77.38	<code>basic_stream_socket::message_out_of_band</code>	638
5.77.39	<code>basic_stream_socket::message_peek</code>	639
5.77.40	<code>basic_stream_socket::native_handle</code>	639
5.77.41	<code>basic_stream_socket::native_handle_type</code>	639
5.77.42	<code>basic_stream_socket::native_non_blocking</code>	639
5.77.42.1	<code>basic_stream_socket::native_non_blocking</code> (1 of 3 overloads)	639
5.77.42.2	<code>basic_stream_socket::native_non_blocking</code> (2 of 3 overloads)	641
5.77.42.3	<code>basic_stream_socket::native_non_blocking</code> (3 of 3 overloads)	642
5.77.43	<code>basic_stream_socket::non_blocking</code>	644
5.77.43.1	<code>basic_stream_socket::non_blocking</code> (1 of 3 overloads)	644
5.77.43.2	<code>basic_stream_socket::non_blocking</code> (2 of 3 overloads)	645
5.77.43.3	<code>basic_stream_socket::non_blocking</code> (3 of 3 overloads)	645
5.77.44	<code>basic_stream_socket::open</code>	646
5.77.44.1	<code>basic_stream_socket::open</code> (1 of 2 overloads)	646
5.77.44.2	<code>basic_stream_socket::open</code> (2 of 2 overloads)	646
5.77.45	<code>basic_stream_socket::operator=</code>	647
5.77.45.1	<code>basic_stream_socket::operator=</code> (1 of 2 overloads)	647
5.77.45.2	<code>basic_stream_socket::operator=</code> (2 of 2 overloads)	647
5.77.46	<code>basic_stream_socket::out_of_band_inline</code>	648
5.77.47	<code>basic_stream_socket::protocol_type</code>	648
5.77.48	<code>basic_stream_socket::read_some</code>	649

5.77.48.1	<code>basic_stream_socket::read_some</code> (1 of 2 overloads)	649
5.77.48.2	<code>basic_stream_socket::read_some</code> (2 of 2 overloads)	650
5.77.49	<code>basic_stream_socket::receive</code>	650
5.77.49.1	<code>basic_stream_socket::receive</code> (1 of 3 overloads)	651
5.77.49.2	<code>basic_stream_socket::receive</code> (2 of 3 overloads)	651
5.77.49.3	<code>basic_stream_socket::receive</code> (3 of 3 overloads)	652
5.77.50	<code>basic_stream_socket::receive_buffer_size</code>	653
5.77.51	<code>basic_stream_socket::receive_low_watermark</code>	653
5.77.52	<code>basic_stream_socket::release</code>	654
5.77.52.1	<code>basic_stream_socket::release</code> (1 of 2 overloads)	654
5.77.52.2	<code>basic_stream_socket::release</code> (2 of 2 overloads)	654
5.77.53	<code>basic_stream_socket::remote_endpoint</code>	655
5.77.53.1	<code>basic_stream_socket::remote_endpoint</code> (1 of 2 overloads)	655
5.77.53.2	<code>basic_stream_socket::remote_endpoint</code> (2 of 2 overloads)	655
5.77.54	<code>basic_stream_socket::reuse_address</code>	656
5.77.55	<code>basic_stream_socket::send</code>	657
5.77.55.1	<code>basic_stream_socket::send</code> (1 of 3 overloads)	657
5.77.55.2	<code>basic_stream_socket::send</code> (2 of 3 overloads)	658
5.77.55.3	<code>basic_stream_socket::send</code> (3 of 3 overloads)	659
5.77.56	<code>basic_stream_socket::send_buffer_size</code>	659
5.77.57	<code>basic_stream_socket::send_low_watermark</code>	660
5.77.58	<code>basic_stream_socket::set_option</code>	660
5.77.58.1	<code>basic_stream_socket::set_option</code> (1 of 2 overloads)	661
5.77.58.2	<code>basic_stream_socket::set_option</code> (2 of 2 overloads)	661
5.77.59	<code>basic_stream_socket::shutdown</code>	662
5.77.59.1	<code>basic_stream_socket::shutdown</code> (1 of 2 overloads)	662
5.77.59.2	<code>basic_stream_socket::shutdown</code> (2 of 2 overloads)	663
5.77.60	<code>basic_stream_socket::shutdown_type</code>	663
5.77.61	<code>basic_stream_socket::wait</code>	663
5.77.61.1	<code>basic_stream_socket::wait</code> (1 of 2 overloads)	664
5.77.61.2	<code>basic_stream_socket::wait</code> (2 of 2 overloads)	664
5.77.62	<code>basic_stream_socket::wait_type</code>	664
5.77.63	<code>basic_stream_socket::write_some</code>	665
5.77.63.1	<code>basic_stream_socket::write_some</code> (1 of 2 overloads)	665
5.77.63.2	<code>basic_stream_socket::write_some</code> (2 of 2 overloads)	666
5.77.64	<code>basic_stream_socket::~basic_stream_socket</code>	666
5.78	<code>basic_streampbuf</code>	666
5.78.1	<code>basic_streampbuf::basic_streampbuf</code>	668
5.78.2	<code>basic_streampbuf::capacity</code>	669

5.78.3	<code>basic_streambuf::commit</code>	669
5.78.4	<code>basic_streambuf::const_buffers_type</code>	669
5.78.5	<code>basic_streambuf::consume</code>	669
5.78.6	<code>basic_streambuf::data</code>	670
5.78.7	<code>basic_streambuf::max_size</code>	670
5.78.8	<code>basic_streambuf::mutable_buffers_type</code>	670
5.78.9	<code>basic_streambuf::overflow</code>	670
5.78.10	<code>basic_streambuf::prepare</code>	670
5.78.11	<code>basic_streambuf::reserve</code>	671
5.78.12	<code>basic_streambuf::size</code>	671
5.78.13	<code>basic_streambuf::underflow</code>	671
5.79	<code>basic_streambuf_ref</code>	671
5.79.1	<code>basic_streambuf_ref::basic_streambuf_ref</code>	672
5.79.1.1	<code>basic_streambuf_ref::basic_streambuf_ref</code> (1 of 3 overloads)	673
5.79.1.2	<code>basic_streambuf_ref::basic_streambuf_ref</code> (2 of 3 overloads)	673
5.79.1.3	<code>basic_streambuf_ref::basic_streambuf_ref</code> (3 of 3 overloads)	673
5.79.2	<code>basic_streambuf_ref::capacity</code>	673
5.79.3	<code>basic_streambuf_ref::commit</code>	673
5.79.4	<code>basic_streambuf_ref::const_buffers_type</code>	673
5.79.5	<code>basic_streambuf_ref::consume</code>	675
5.79.6	<code>basic_streambuf_ref::data</code>	675
5.79.7	<code>basic_streambuf_ref::max_size</code>	675
5.79.8	<code>basic_streambuf_ref::mutable_buffers_type</code>	676
5.79.9	<code>basic_streambuf_ref::prepare</code>	678
5.79.10	<code>basic_streambuf_ref::size</code>	678
5.80	<code>basic_waitable_timer</code>	678
5.80.1	<code>basic_waitable_timer::async_wait</code>	681
5.80.2	<code>basic_waitable_timer::basic_waitable_timer</code>	681
5.80.2.1	<code>basic_waitable_timer::basic_waitable_timer</code> (1 of 4 overloads)	682
5.80.2.2	<code>basic_waitable_timer::basic_waitable_timer</code> (2 of 4 overloads)	682
5.80.2.3	<code>basic_waitable_timer::basic_waitable_timer</code> (3 of 4 overloads)	682
5.80.2.4	<code>basic_waitable_timer::basic_waitable_timer</code> (4 of 4 overloads)	683
5.80.3	<code>basic_waitable_timer::cancel</code>	683
5.80.3.1	<code>basic_waitable_timer::cancel</code> (1 of 2 overloads)	683
5.80.3.2	<code>basic_waitable_timer::cancel</code> (2 of 2 overloads)	684
5.80.4	<code>basic_waitable_timer::cancel_one</code>	684
5.80.4.1	<code>basic_waitable_timer::cancel_one</code> (1 of 2 overloads)	685
5.80.4.2	<code>basic_waitable_timer::cancel_one</code> (2 of 2 overloads)	685
5.80.5	<code>basic_waitable_timer::clock_type</code>	686

5.80.6	<code>basic_waitable_timer::duration</code>	686
5.80.7	<code>basic_waitable_timer::executor_type</code>	686
5.80.8	<code>basic_waitable_timer::expires_after</code>	687
5.80.9	<code>basic_waitable_timer::expires_at</code>	688
5.80.9.1	<code>basic_waitable_timer::expires_at</code> (1 of 3 overloads)	688
5.80.9.2	<code>basic_waitable_timer::expires_at</code> (2 of 3 overloads)	688
5.80.9.3	<code>basic_waitable_timer::expires_at</code> (3 of 3 overloads)	689
5.80.10	<code>basic_waitable_timer::expires_from_now</code>	690
5.80.10.1	<code>basic_waitable_timer::expires_from_now</code> (1 of 3 overloads)	690
5.80.10.2	<code>basic_waitable_timer::expires_from_now</code> (2 of 3 overloads)	690
5.80.10.3	<code>basic_waitable_timer::expires_from_now</code> (3 of 3 overloads)	691
5.80.11	<code>basic_waitable_timer::expiry</code>	691
5.80.12	<code>basic_waitable_timer::get_executor</code>	691
5.80.13	<code>basic_waitable_timer::get_io_context</code>	691
5.80.14	<code>basic_waitable_timer::get_io_service</code>	692
5.80.15	<code>basic_waitable_timer::operator=</code>	692
5.80.16	<code>basic_waitable_timer::time_point</code>	692
5.80.17	<code>basic_waitable_timer::traits_type</code>	692
5.80.18	<code>basic_waitable_timer::wait</code>	693
5.80.18.1	<code>basic_waitable_timer::wait</code> (1 of 2 overloads)	693
5.80.18.2	<code>basic_waitable_timer::wait</code> (2 of 2 overloads)	693
5.80.19	<code>basic_waitable_timer::~basic_waitable_timer</code>	693
5.81	<code>basic_yield_context</code>	694
5.81.1	<code>basic_yield_context::basic_yield_context</code>	694
5.81.1.1	<code>basic_yield_context::basic_yield_context</code> (1 of 2 overloads)	695
5.81.1.2	<code>basic_yield_context::basic_yield_context</code> (2 of 2 overloads)	695
5.81.2	<code>basic_yield_context::callee_type</code>	695
5.81.3	<code>basic_yield_context::caller_type</code>	696
5.81.4	<code>basic_yield_context::operator[]</code>	696
5.82	<code>bind_executor</code>	696
5.82.1	<code>bind_executor</code> (1 of 2 overloads)	697
5.82.2	<code>bind_executor</code> (2 of 2 overloads)	697
5.83	<code>buffer</code>	697
5.83.1	<code>buffer</code> (1 of 32 overloads)	703
5.83.2	<code>buffer</code> (2 of 32 overloads)	704
5.83.3	<code>buffer</code> (3 of 32 overloads)	704
5.83.4	<code>buffer</code> (4 of 32 overloads)	704
5.83.5	<code>buffer</code> (5 of 32 overloads)	704
5.83.6	<code>buffer</code> (6 of 32 overloads)	705

5.83.7	buffer (7 of 32 overloads)	705
5.83.8	buffer (8 of 32 overloads)	705
5.83.9	buffer (9 of 32 overloads)	706
5.83.10	buffer (10 of 32 overloads)	706
5.83.11	buffer (11 of 32 overloads)	706
5.83.12	buffer (12 of 32 overloads)	707
5.83.13	buffer (13 of 32 overloads)	707
5.83.14	buffer (14 of 32 overloads)	707
5.83.15	buffer (15 of 32 overloads)	708
5.83.16	buffer (16 of 32 overloads)	708
5.83.17	buffer (17 of 32 overloads)	708
5.83.18	buffer (18 of 32 overloads)	709
5.83.19	buffer (19 of 32 overloads)	709
5.83.20	buffer (20 of 32 overloads)	709
5.83.21	buffer (21 of 32 overloads)	710
5.83.22	buffer (22 of 32 overloads)	710
5.83.23	buffer (23 of 32 overloads)	710
5.83.24	buffer (24 of 32 overloads)	711
5.83.25	buffer (25 of 32 overloads)	711
5.83.26	buffer (26 of 32 overloads)	712
5.83.27	buffer (27 of 32 overloads)	712
5.83.28	buffer (28 of 32 overloads)	712
5.83.29	buffer (29 of 32 overloads)	713
5.83.30	buffer (30 of 32 overloads)	713
5.83.31	buffer (31 of 32 overloads)	714
5.83.32	buffer (32 of 32 overloads)	714
5.84	buffer_cast	714
5.84.1	buffer_cast (1 of 2 overloads)	715
5.84.2	buffer_cast (2 of 2 overloads)	715
5.85	buffer_copy	715
5.85.1	buffer_copy (1 of 2 overloads)	716
5.85.2	buffer_copy (2 of 2 overloads)	717
5.86	buffer_sequence_begin	717
5.86.1	buffer_sequence_begin (1 of 4 overloads)	718
5.86.2	buffer_sequence_begin (2 of 4 overloads)	718
5.86.3	buffer_sequence_begin (3 of 4 overloads)	718
5.86.4	buffer_sequence_begin (4 of 4 overloads)	718
5.87	buffer_sequence_end	718
5.87.1	buffer_sequence_end (1 of 4 overloads)	719

5.87.2	buffer_sequence_end (2 of 4 overloads)	719
5.87.3	buffer_sequence_end (3 of 4 overloads)	719
5.87.4	buffer_sequence_end (4 of 4 overloads)	719
5.88	buffer_size	719
5.89	buffered_read_stream	720
5.89.1	buffered_read_stream::async_fill	722
5.89.2	buffered_read_stream::async_read_some	722
5.89.3	buffered_read_stream::async_write_some	722
5.89.4	buffered_read_stream::buffered_read_stream	722
5.89.4.1	buffered_read_stream::buffered_read_stream (1 of 2 overloads)	723
5.89.4.2	buffered_read_stream::buffered_read_stream (2 of 2 overloads)	723
5.89.5	buffered_read_stream::close	723
5.89.5.1	buffered_read_stream::close (1 of 2 overloads)	723
5.89.5.2	buffered_read_stream::close (2 of 2 overloads)	723
5.89.6	buffered_read_stream::default_buffer_size	723
5.89.7	buffered_read_stream::executor_type	723
5.89.8	buffered_read_stream::fill	724
5.89.8.1	buffered_read_stream::fill (1 of 2 overloads)	724
5.89.8.2	buffered_read_stream::fill (2 of 2 overloads)	724
5.89.9	buffered_read_stream::get_executor	724
5.89.10	buffered_read_stream::get_io_context	724
5.89.11	buffered_read_stream::get_io_service	724
5.89.12	buffered_read_stream::in_avail	725
5.89.12.1	buffered_read_stream::in_avail (1 of 2 overloads)	725
5.89.12.2	buffered_read_stream::in_avail (2 of 2 overloads)	725
5.89.13	buffered_read_stream::lowest_layer	725
5.89.13.1	buffered_read_stream::lowest_layer (1 of 2 overloads)	725
5.89.13.2	buffered_read_stream::lowest_layer (2 of 2 overloads)	725
5.89.14	buffered_read_stream::lowest_layer_type	725
5.89.15	buffered_read_stream::next_layer	726
5.89.16	buffered_read_stream::next_layer_type	726
5.89.17	buffered_read_stream::peek	726
5.89.17.1	buffered_read_stream::peek (1 of 2 overloads)	726
5.89.17.2	buffered_read_stream::peek (2 of 2 overloads)	727
5.89.18	buffered_read_stream::read_some	727
5.89.18.1	buffered_read_stream::read_some (1 of 2 overloads)	727
5.89.18.2	buffered_read_stream::read_some (2 of 2 overloads)	727
5.89.19	buffered_read_stream::write_some	727
5.89.19.1	buffered_read_stream::write_some (1 of 2 overloads)	728

5.89.19.2	buffered_read_stream::write_some (2 of 2 overloads)	728
5.90	buffered_stream	728
5.90.1	buffered_stream::async_fill	730
5.90.2	buffered_stream::async_flush	730
5.90.3	buffered_stream::async_read_some	730
5.90.4	buffered_stream::async_write_some	731
5.90.5	buffered_stream::buffered_stream	731
5.90.5.1	buffered_stream::buffered_stream (1 of 2 overloads)	731
5.90.5.2	buffered_stream::buffered_stream (2 of 2 overloads)	731
5.90.6	buffered_stream::close	731
5.90.6.1	buffered_stream::close (1 of 2 overloads)	732
5.90.6.2	buffered_stream::close (2 of 2 overloads)	732
5.90.7	buffered_stream::executor_type	732
5.90.8	buffered_stream::fill	732
5.90.8.1	buffered_stream::fill (1 of 2 overloads)	732
5.90.8.2	buffered_stream::fill (2 of 2 overloads)	732
5.90.9	buffered_stream::flush	733
5.90.9.1	buffered_stream::flush (1 of 2 overloads)	733
5.90.9.2	buffered_stream::flush (2 of 2 overloads)	733
5.90.10	buffered_stream::get_executor	733
5.90.11	buffered_stream::get_io_context	733
5.90.12	buffered_stream::get_io_service	733
5.90.13	buffered_stream::in_avail	734
5.90.13.1	buffered_stream::in_avail (1 of 2 overloads)	734
5.90.13.2	buffered_stream::in_avail (2 of 2 overloads)	734
5.90.14	buffered_stream::lowest_layer	734
5.90.14.1	buffered_stream::lowest_layer (1 of 2 overloads)	734
5.90.14.2	buffered_stream::lowest_layer (2 of 2 overloads)	734
5.90.15	buffered_stream::lowest_layer_type	734
5.90.16	buffered_stream::next_layer	735
5.90.17	buffered_stream::next_layer_type	735
5.90.18	buffered_stream::peek	735
5.90.18.1	buffered_stream::peek (1 of 2 overloads)	735
5.90.18.2	buffered_stream::peek (2 of 2 overloads)	736
5.90.19	buffered_stream::read_some	736
5.90.19.1	buffered_stream::read_some (1 of 2 overloads)	736
5.90.19.2	buffered_stream::read_some (2 of 2 overloads)	736
5.90.20	buffered_stream::write_some	736
5.90.20.1	buffered_stream::write_some (1 of 2 overloads)	737

5.90.20.2	<code>buffered_stream::write_some</code> (2 of 2 overloads)	737
5.91	<code>buffered_write_stream</code>	737
5.91.1	<code>buffered_write_stream::async_flush</code>	739
5.91.2	<code>buffered_write_stream::async_read_some</code>	739
5.91.3	<code>buffered_write_stream::async_write_some</code>	739
5.91.4	<code>buffered_write_stream::buffered_write_stream</code>	740
5.91.4.1	<code>buffered_write_stream::buffered_write_stream</code> (1 of 2 overloads)	740
5.91.4.2	<code>buffered_write_stream::buffered_write_stream</code> (2 of 2 overloads)	740
5.91.5	<code>buffered_write_stream::close</code>	740
5.91.5.1	<code>buffered_write_stream::close</code> (1 of 2 overloads)	740
5.91.5.2	<code>buffered_write_stream::close</code> (2 of 2 overloads)	740
5.91.6	<code>buffered_write_stream::default_buffer_size</code>	741
5.91.7	<code>buffered_write_stream::executor_type</code>	741
5.91.8	<code>buffered_write_stream::flush</code>	741
5.91.8.1	<code>buffered_write_stream::flush</code> (1 of 2 overloads)	741
5.91.8.2	<code>buffered_write_stream::flush</code> (2 of 2 overloads)	741
5.91.9	<code>buffered_write_stream::get_executor</code>	741
5.91.10	<code>buffered_write_stream::get_io_context</code>	742
5.91.11	<code>buffered_write_stream::get_io_service</code>	742
5.91.12	<code>buffered_write_stream::in_avail</code>	742
5.91.12.1	<code>buffered_write_stream::in_avail</code> (1 of 2 overloads)	742
5.91.12.2	<code>buffered_write_stream::in_avail</code> (2 of 2 overloads)	742
5.91.13	<code>buffered_write_stream::lowest_layer</code>	742
5.91.13.1	<code>buffered_write_stream::lowest_layer</code> (1 of 2 overloads)	742
5.91.13.2	<code>buffered_write_stream::lowest_layer</code> (2 of 2 overloads)	743
5.91.14	<code>buffered_write_stream::lowest_layer_type</code>	743
5.91.15	<code>buffered_write_stream::next_layer</code>	743
5.91.16	<code>buffered_write_stream::next_layer_type</code>	743
5.91.17	<code>buffered_write_stream::peek</code>	743
5.91.17.1	<code>buffered_write_stream::peek</code> (1 of 2 overloads)	744
5.91.17.2	<code>buffered_write_stream::peek</code> (2 of 2 overloads)	744
5.91.18	<code>buffered_write_stream::read_some</code>	744
5.91.18.1	<code>buffered_write_stream::read_some</code> (1 of 2 overloads)	744
5.91.18.2	<code>buffered_write_stream::read_some</code> (2 of 2 overloads)	744
5.91.19	<code>buffered_write_stream::write_some</code>	745
5.91.19.1	<code>buffered_write_stream::write_some</code> (1 of 2 overloads)	745
5.91.19.2	<code>buffered_write_stream::write_some</code> (2 of 2 overloads)	745
5.92	<code>buffers_begin</code>	745
5.93	<code>buffers_end</code>	746

5.94	buffers_iterator	746
5.94.1	buffers_iterator::begin	747
5.94.2	buffers_iterator::buffers_iterator	747
5.94.3	buffers_iterator::difference_type	748
5.94.4	buffers_iterator::end	748
5.94.5	buffers_iterator::iterator_category	748
5.94.6	buffers_iterator::operator *	748
5.94.7	buffers_iterator::operator!=	748
5.94.8	buffers_iterator::operator+	749
5.94.8.1	buffers_iterator::operator+ (1 of 2 overloads)	749
5.94.8.2	buffers_iterator::operator+ (2 of 2 overloads)	749
5.94.9	buffers_iterator::operator++	749
5.94.9.1	buffers_iterator::operator++ (1 of 2 overloads)	750
5.94.9.2	buffers_iterator::operator++ (2 of 2 overloads)	750
5.94.10	buffers_iterator::operator+=	750
5.94.11	buffers_iterator::operator-	750
5.94.11.1	buffers_iterator::operator- (1 of 2 overloads)	750
5.94.11.2	buffers_iterator::operator- (2 of 2 overloads)	750
5.94.12	buffers_iterator::operator--	751
5.94.12.1	buffers_iterator::operator-- (1 of 2 overloads)	751
5.94.12.2	buffers_iterator::operator-- (2 of 2 overloads)	751
5.94.13	buffers_iterator::operator-=	751
5.94.14	buffers_iterator::operator->	751
5.94.15	buffers_iterator::operator<	751
5.94.16	buffers_iterator::operator<=	752
5.94.17	buffers_iterator::operator==	752
5.94.18	buffers_iterator::operator>	752
5.94.19	buffers_iterator::operator>=	752
5.94.20	buffers_iterator::operator[]	753
5.94.21	buffers_iterator::pointer	753
5.94.22	buffers_iterator::reference	753
5.94.23	buffers_iterator::value_type	753
5.95	connect	754
5.95.1	connect (1 of 12 overloads)	756
5.95.2	connect (2 of 12 overloads)	757
5.95.3	connect (3 of 12 overloads)	757
5.95.4	connect (4 of 12 overloads)	758
5.95.5	connect (5 of 12 overloads)	759
5.95.6	connect (6 of 12 overloads)	759

5.95.7	connect (7 of 12 overloads)	760
5.95.8	connect (8 of 12 overloads)	761
5.95.9	connect (9 of 12 overloads)	763
5.95.10	connect (10 of 12 overloads)	763
5.95.11	connect (11 of 12 overloads)	764
5.95.12	connect (12 of 12 overloads)	766
5.96	const_buffer	767
5.96.1	const_buffer::const_buffer	768
5.96.1.1	const_buffer::const_buffer (1 of 3 overloads)	768
5.96.1.2	const_buffer::const_buffer (2 of 3 overloads)	768
5.96.1.3	const_buffer::const_buffer (3 of 3 overloads)	768
5.96.2	const_buffer::data	769
5.96.3	const_buffer::operator+	769
5.96.3.1	const_buffer::operator+ (1 of 2 overloads)	769
5.96.3.2	const_buffer::operator+ (2 of 2 overloads)	769
5.96.4	const_buffer::operator+=	769
5.96.5	const_buffer::size	769
5.97	const_buffers_1	769
5.97.1	const_buffers_1::begin	770
5.97.2	const_buffers_1::const_buffers_1	770
5.97.2.1	const_buffers_1::const_buffers_1 (1 of 2 overloads)	771
5.97.2.2	const_buffers_1::const_buffers_1 (2 of 2 overloads)	771
5.97.3	const_buffers_1::const_iterator	771
5.97.4	const_buffers_1::data	771
5.97.5	const_buffers_1::end	771
5.97.6	const_buffers_1::operator+	772
5.97.6.1	const_buffers_1::operator+ (1 of 2 overloads)	772
5.97.6.2	const_buffers_1::operator+ (2 of 2 overloads)	772
5.97.7	const_buffers_1::operator+=	772
5.97.8	const_buffers_1::size	772
5.97.9	const_buffers_1::value_type	772
5.98	coroutine	773
5.98.1	coroutine::coroutine	777
5.98.2	coroutine::is_child	777
5.98.3	coroutine::is_complete	777
5.98.4	coroutine::is_parent	778
5.99	deadline_timer	778
5.100	defer	780
5.100.1	defer (1 of 3 overloads)	781

5.100.2	defer (2 of 3 overloads)	781
5.100.3	defer (3 of 3 overloads)	782
5.101	dispatch	782
5.101.1	dispatch (1 of 3 overloads)	783
5.101.2	dispatch (2 of 3 overloads)	783
5.101.3	dispatch (3 of 3 overloads)	784
5.102	dynamic_buffer	784
5.102.1	dynamic_buffer (1 of 4 overloads)	785
5.102.2	dynamic_buffer (2 of 4 overloads)	785
5.102.3	dynamic_buffer (3 of 4 overloads)	785
5.102.4	dynamic_buffer (4 of 4 overloads)	785
5.103	dynamic_string_buffer	786
5.103.1	dynamic_string_buffer::capacity	787
5.103.2	dynamic_string_buffer::commit	787
5.103.3	dynamic_string_buffer::const_buffers_type	787
5.103.4	dynamic_string_buffer::consume	788
5.103.5	dynamic_string_buffer::data	788
5.103.6	dynamic_string_buffer::dynamic_string_buffer	789
5.103.6.1	dynamic_string_buffer::dynamic_string_buffer (1 of 2 overloads)	789
5.103.6.2	dynamic_string_buffer::dynamic_string_buffer (2 of 2 overloads)	789
5.103.7	dynamic_string_buffer::max_size	789
5.103.8	dynamic_string_buffer::mutable_buffers_type	790
5.103.9	dynamic_string_buffer::prepare	790
5.103.10	dynamic_string_buffer::size	791
5.104	dynamic_vector_buffer	791
5.104.1	dynamic_vector_buffer::capacity	792
5.104.2	dynamic_vector_buffer::commit	792
5.104.3	dynamic_vector_buffer::const_buffers_type	793
5.104.4	dynamic_vector_buffer::consume	793
5.104.5	dynamic_vector_buffer::data	794
5.104.6	dynamic_vector_buffer::dynamic_vector_buffer	794
5.104.6.1	dynamic_vector_buffer::dynamic_vector_buffer (1 of 2 overloads)	794
5.104.6.2	dynamic_vector_buffer::dynamic_vector_buffer (2 of 2 overloads)	795
5.104.7	dynamic_vector_buffer::max_size	795
5.104.8	dynamic_vector_buffer::mutable_buffers_type	795
5.104.9	dynamic_vector_buffer::prepare	796
5.104.10	dynamic_vector_buffer::size	796
5.105	error::addrinfo_category	796
5.106	error::addrinfo_errors	797

5.107	error::basic_errors	797
5.108	error::get_addrinfo_category	798
5.109	error::get_misc_category	798
5.110	error::get_netdb_category	799
5.111	error::get_ssl_category	799
5.112	error::get_system_category	799
5.113	error::make_error_code	799
5.113.1	error::make_error_code (1 of 5 overloads)	800
5.113.2	error::make_error_code (2 of 5 overloads)	800
5.113.3	error::make_error_code (3 of 5 overloads)	800
5.113.4	error::make_error_code (4 of 5 overloads)	800
5.113.5	error::make_error_code (5 of 5 overloads)	800
5.114	error::misc_category	800
5.115	error::misc_errors	800
5.116	error::netdb_category	801
5.117	error::netdb_errors	801
5.118	error::ssl_category	801
5.119	error::ssl_errors	802
5.120	error::system_category	802
5.121	error_category	802
5.121.1	error_category::message	803
5.121.2	error_category::name	803
5.121.3	error_category::operator!=	803
5.121.4	error_category::operator==	803
5.121.5	error_category::~error_category	803
5.122	error_code	803
5.122.1	error_code::assign	804
5.122.2	error_code::category	804
5.122.3	error_code::clear	805
5.122.4	error_code::error_code	805
5.122.4.1	error_code::error_code (1 of 3 overloads)	805
5.122.4.2	error_code::error_code (2 of 3 overloads)	805
5.122.4.3	error_code::error_code (3 of 3 overloads)	805
5.122.5	error_code::message	805
5.122.6	error_code::operator unspecified_bool_type	806
5.122.7	error_code::operator!	806
5.122.8	error_code::operator!=	806
5.122.9	error_code::operator==	806
5.122.10	error_code::unspecified_bool_true	806

5.122.11	error_code::unspecified_bool_type	806
5.122.12	error_code::value	807
5.123	error_code::unspecified_bool_type_t	807
5.124	execution_context	807
5.124.1	execution_context::add_service	809
5.124.2	execution_context::destroy	810
5.124.3	execution_context::execution_context	810
5.124.4	execution_context::fork_event	810
5.124.5	execution_context::has_service	810
5.124.6	execution_context::make_service	811
5.124.7	execution_context::notify_fork	811
5.124.8	execution_context::shutdown	812
5.124.9	execution_context::use_service	812
5.124.9.1	execution_context::use_service (1 of 2 overloads)	813
5.124.9.2	execution_context::use_service (2 of 2 overloads)	813
5.124.10	execution_context::~execution_context	814
5.125	execution_context::id	814
5.125.1	execution_context::id::id	814
5.126	execution_context::service	814
5.126.1	execution_context::service::context	815
5.126.2	execution_context::service::service	815
5.126.3	execution_context::service::~service	815
5.126.4	execution_context::service::notify_fork	816
5.126.5	execution_context::service::shutdown	816
5.127	executor	816
5.127.1	executor::context	817
5.127.2	executor::defer	817
5.127.3	executor::dispatch	818
5.127.4	executor::executor	818
5.127.4.1	executor::executor (1 of 6 overloads)	819
5.127.4.2	executor::executor (2 of 6 overloads)	819
5.127.4.3	executor::executor (3 of 6 overloads)	819
5.127.4.4	executor::executor (4 of 6 overloads)	819
5.127.4.5	executor::executor (5 of 6 overloads)	819
5.127.4.6	executor::executor (6 of 6 overloads)	820
5.127.5	executor::on_work_finished	820
5.127.6	executor::on_work_started	820
5.127.7	executor::operator unspecified_bool_type	820
5.127.8	executor::operator!=	820

5.127.9	executor::operator=	820
5.127.9.1	executor::operator= (1 of 4 overloads)	821
5.127.9.2	executor::operator= (2 of 4 overloads)	821
5.127.9.3	executor::operator= (3 of 4 overloads)	821
5.127.9.4	executor::operator= (4 of 4 overloads)	821
5.127.10	executor::operator==	821
5.127.11	executor::post	822
5.127.12	executor::target	822
5.127.12.1	executor::target (1 of 2 overloads)	822
5.127.12.2	executor::target (2 of 2 overloads)	822
5.127.13	executor::target_type	823
5.127.14	executor::unspecified_bool_true	823
5.127.15	executor::unspecified_bool_type	823
5.127.16	executor::~executor	823
5.128	executor::unspecified_bool_type_t	823
5.129	executor_arg	823
5.130	executor_arg_t	824
5.130.1	executor_arg_t::executor_arg_t	824
5.131	executor_binder	824
5.131.1	executor_binder::argument_type	825
5.131.2	executor_binder::executor_binder	826
5.131.2.1	executor_binder::executor_binder (1 of 9 overloads)	827
5.131.2.2	executor_binder::executor_binder (2 of 9 overloads)	827
5.131.2.3	executor_binder::executor_binder (3 of 9 overloads)	827
5.131.2.4	executor_binder::executor_binder (4 of 9 overloads)	828
5.131.2.5	executor_binder::executor_binder (5 of 9 overloads)	828
5.131.2.6	executor_binder::executor_binder (6 of 9 overloads)	828
5.131.2.7	executor_binder::executor_binder (7 of 9 overloads)	828
5.131.2.8	executor_binder::executor_binder (8 of 9 overloads)	828
5.131.2.9	executor_binder::executor_binder (9 of 9 overloads)	829
5.131.3	executor_binder::executor_type	829
5.131.4	executor_binder::first_argument_type	829
5.131.5	executor_binder::get	829
5.131.5.1	executor_binder::get (1 of 2 overloads)	830
5.131.5.2	executor_binder::get (2 of 2 overloads)	830
5.131.6	executor_binder::get_executor	830
5.131.7	executor_binder::operator()	830
5.131.7.1	executor_binder::operator() (1 of 2 overloads)	830
5.131.7.2	executor_binder::operator() (2 of 2 overloads)	830

5.131.8	executor_binder::result_type	830
5.131.9	executor_binder::second_argument_type	831
5.131.10	executor_binder::target_type	831
5.131.11	executor_binder::~executor_binder	831
5.132	executor_work_guard	831
5.132.1	executor_work_guard::executor_type	832
5.132.2	executor_work_guard::executor_work_guard	832
5.132.2.1	executor_work_guard::executor_work_guard (1 of 3 overloads)	833
5.132.2.2	executor_work_guard::executor_work_guard (2 of 3 overloads)	833
5.132.2.3	executor_work_guard::executor_work_guard (3 of 3 overloads)	833
5.132.3	executor_work_guard::get_executor	833
5.132.4	executor_work_guard::owns_work	833
5.132.5	executor_work_guard::reset	833
5.132.6	executor_work_guard::~executor_work_guard	833
5.133	generic::basic_endpoint	834
5.133.1	generic::basic_endpoint::basic_endpoint	835
5.133.1.1	generic::basic_endpoint::basic_endpoint (1 of 4 overloads)	835
5.133.1.2	generic::basic_endpoint::basic_endpoint (2 of 4 overloads)	836
5.133.1.3	generic::basic_endpoint::basic_endpoint (3 of 4 overloads)	836
5.133.1.4	generic::basic_endpoint::basic_endpoint (4 of 4 overloads)	836
5.133.2	generic::basic_endpoint::capacity	836
5.133.3	generic::basic_endpoint::data	836
5.133.3.1	generic::basic_endpoint::data (1 of 2 overloads)	836
5.133.3.2	generic::basic_endpoint::data (2 of 2 overloads)	836
5.133.4	generic::basic_endpoint::data_type	837
5.133.5	generic::basic_endpoint::operator!=	837
5.133.6	generic::basic_endpoint::operator<	837
5.133.7	generic::basic_endpoint::operator<=	837
5.133.8	generic::basic_endpoint::operator=	838
5.133.9	generic::basic_endpoint::operator==	838
5.133.10	generic::basic_endpoint::operator>	838
5.133.11	generic::basic_endpoint::operator>=	838
5.133.12	generic::basic_endpoint::protocol	838
5.133.13	generic::basic_endpoint::protocol_type	839
5.133.14	generic::basic_endpoint::resize	839
5.133.15	generic::basic_endpoint::size	839
5.134	generic::datagram_protocol	839
5.134.1	generic::datagram_protocol::datagram_protocol	840
5.134.1.1	generic::datagram_protocol::datagram_protocol (1 of 2 overloads)	841

5.134.1.2	generic::datagram_protocol::datagram_protocol (2 of 2 overloads)	841
5.134.2	generic::datagram_protocol::endpoint	841
5.134.3	generic::datagram_protocol::family	842
5.134.4	generic::datagram_protocol::operator!=	843
5.134.5	generic::datagram_protocol::operator==	843
5.134.6	generic::datagram_protocol::protocol	843
5.134.7	generic::datagram_protocol::socket	843
5.134.8	generic::datagram_protocol::type	847
5.135	generic::raw_protocol	847
5.135.1	generic::raw_protocol::endpoint	848
5.135.2	generic::raw_protocol::family	850
5.135.3	generic::raw_protocol::operator!=	850
5.135.4	generic::raw_protocol::operator==	850
5.135.5	generic::raw_protocol::protocol	850
5.135.6	generic::raw_protocol::raw_protocol	850
5.135.6.1	generic::raw_protocol::raw_protocol (1 of 2 overloads)	851
5.135.6.2	generic::raw_protocol::raw_protocol (2 of 2 overloads)	851
5.135.7	generic::raw_protocol::socket	851
5.135.8	generic::raw_protocol::type	855
5.136	generic::seq_packet_protocol	855
5.136.1	generic::seq_packet_protocol::endpoint	856
5.136.2	generic::seq_packet_protocol::family	857
5.136.3	generic::seq_packet_protocol::operator!=	857
5.136.4	generic::seq_packet_protocol::operator==	858
5.136.5	generic::seq_packet_protocol::protocol	858
5.136.6	generic::seq_packet_protocol::seq_packet_protocol	858
5.136.6.1	generic::seq_packet_protocol::seq_packet_protocol (1 of 2 overloads)	858
5.136.6.2	generic::seq_packet_protocol::seq_packet_protocol (2 of 2 overloads)	858
5.136.7	generic::seq_packet_protocol::socket	859
5.136.8	generic::seq_packet_protocol::type	862
5.137	generic::stream_protocol	862
5.137.1	generic::stream_protocol::endpoint	863
5.137.2	generic::stream_protocol::family	865
5.137.3	generic::stream_protocol::iostream	865
5.137.4	generic::stream_protocol::operator!=	866
5.137.5	generic::stream_protocol::operator==	867
5.137.6	generic::stream_protocol::protocol	867
5.137.7	generic::stream_protocol::socket	867
5.137.8	generic::stream_protocol::stream_protocol	871

5.137.8.1	generic::stream_protocol::stream_protocol (1 of 2 overloads)	871
5.137.8.2	generic::stream_protocol::stream_protocol (2 of 2 overloads)	871
5.137.9	generic::stream_protocol::type	871
5.138	get_associated_allocator	871
5.138.1	get_associated_allocator (1 of 2 overloads)	872
5.138.2	get_associated_allocator (2 of 2 overloads)	872
5.139	get_associated_executor	872
5.139.1	get_associated_executor (1 of 3 overloads)	873
5.139.2	get_associated_executor (2 of 3 overloads)	873
5.139.3	get_associated_executor (3 of 3 overloads)	874
5.140	handler_type	874
5.140.1	handler_type::type	874
5.141	has_service	875
5.142	high_resolution_timer	875
5.143	invalid_service_owner	878
5.143.1	invalid_service_owner::invalid_service_owner	878
5.144	io_context	878
5.144.1	io_context::add_service	883
5.144.2	io_context::count_type	883
5.144.3	io_context::destroy	884
5.144.4	io_context::dispatch	884
5.144.5	io_context::fork_event	884
5.144.6	io_context::get_executor	885
5.144.7	io_context::has_service	885
5.144.8	io_context::io_context	885
5.144.8.1	io_context::io_context (1 of 2 overloads)	886
5.144.8.2	io_context::io_context (2 of 2 overloads)	886
5.144.9	io_context::make_service	886
5.144.10	io_context::notify_fork	887
5.144.11	io_context::poll	887
5.144.11.1	io_context::poll (1 of 2 overloads)	888
5.144.11.2	io_context::poll (2 of 2 overloads)	888
5.144.12	io_context::poll_one	888
5.144.12.1	io_context::poll_one (1 of 2 overloads)	888
5.144.12.2	io_context::poll_one (2 of 2 overloads)	889
5.144.13	io_context::post	889
5.144.14	io_context::reset	890
5.144.15	io_context::restart	890
5.144.16	io_context::run	890

5.144.16.1	io_context::run (1 of 2 overloads)	890
5.144.16.2	io_context::run (2 of 2 overloads)	891
5.144.17	io_context::run_for	891
5.144.18	io_context::run_one	892
5.144.18.1	io_context::run_one (1 of 2 overloads)	892
5.144.18.2	io_context::run_one (2 of 2 overloads)	892
5.144.19	io_context::run_one_for	893
5.144.20	io_context::run_one_until	893
5.144.21	io_context::run_until	894
5.144.22	io_context::shutdown	894
5.144.23	io_context::stop	894
5.144.24	io_context::stopped	894
5.144.25	io_context::use_service	895
5.144.25.1	io_context::use_service (1 of 2 overloads)	895
5.144.25.2	io_context::use_service (2 of 2 overloads)	896
5.144.26	io_context::wrap	896
5.144.27	io_context::~io_context	897
5.145	io_context::executor_type	897
5.145.1	io_context::executor_type::context	898
5.145.2	io_context::executor_type::defer	898
5.145.3	io_context::executor_type::dispatch	899
5.145.4	io_context::executor_type::on_work_finished	899
5.145.5	io_context::executor_type::on_work_started	899
5.145.6	io_context::executor_type::operator!=	900
5.145.7	io_context::executor_type::operator==	900
5.145.8	io_context::executor_type::post	900
5.145.9	io_context::executor_type::running_in_this_thread	901
5.146	io_context::service	901
5.146.1	io_context::service::get_io_context	901
5.146.2	io_context::service::get_io_service	901
5.146.3	io_context::service::service	902
5.146.4	io_context::service::~service	902
5.147	io_context::strand	902
5.147.1	io_context::strand::context	904
5.147.2	io_context::strand::defer	904
5.147.3	io_context::strand::dispatch	904
5.147.3.1	io_context::strand::dispatch (1 of 2 overloads)	905
5.147.3.2	io_context::strand::dispatch (2 of 2 overloads)	905
5.147.4	io_context::strand::get_io_context	906

5.147.5	io_context::strand::get_io_service	906
5.147.6	io_context::strand::on_work_finished	906
5.147.7	io_context::strand::on_work_started	906
5.147.8	io_context::strand::operator!=	906
5.147.9	io_context::strand::operator==	907
5.147.10	io_context::strand::post	907
5.147.10.1	io_context::strand::post (1 of 2 overloads)	907
5.147.10.2	io_context::strand::post (2 of 2 overloads)	908
5.147.11	io_context::strand::running_in_this_thread	908
5.147.12	io_context::strand::strand	908
5.147.13	io_context::strand::wrap	908
5.147.14	io_context::strand::~strand	909
5.148	io_context::work	909
5.148.1	io_context::work::get_io_context	910
5.148.2	io_context::work::get_io_service	910
5.148.3	io_context::work::work	910
5.148.3.1	io_context::work::work (1 of 2 overloads)	910
5.148.3.2	io_context::work::work (2 of 2 overloads)	910
5.148.4	io_context::work::~work	911
5.149	io_service	911
5.150	ip::address	915
5.150.1	ip::address::address	916
5.150.1.1	ip::address::address (1 of 4 overloads)	917
5.150.1.2	ip::address::address (2 of 4 overloads)	917
5.150.1.3	ip::address::address (3 of 4 overloads)	917
5.150.1.4	ip::address::address (4 of 4 overloads)	917
5.150.2	ip::address::from_string	917
5.150.2.1	ip::address::from_string (1 of 4 overloads)	918
5.150.2.2	ip::address::from_string (2 of 4 overloads)	918
5.150.2.3	ip::address::from_string (3 of 4 overloads)	918
5.150.2.4	ip::address::from_string (4 of 4 overloads)	918
5.150.3	ip::address::is_loopback	918
5.150.4	ip::address::is_multicast	919
5.150.5	ip::address::is_unspecified	919
5.150.6	ip::address::is_v4	919
5.150.7	ip::address::is_v6	919
5.150.8	ip::address::make_address	919
5.150.8.1	ip::address::make_address (1 of 6 overloads)	920
5.150.8.2	ip::address::make_address (2 of 6 overloads)	920

5.150.8.3	ip::address::make_address (3 of 6 overloads)	920
5.150.8.4	ip::address::make_address (4 of 6 overloads)	920
5.150.8.5	ip::address::make_address (5 of 6 overloads)	920
5.150.8.6	ip::address::make_address (6 of 6 overloads)	920
5.150.9	ip::address::operator!=	920
5.150.10	ip::address::operator<	921
5.150.11	ip::address::operator<<	921
5.150.12	ip::address::operator<=	921
5.150.13	ip::address::operator=	922
5.150.13.1	ip::address::operator= (1 of 3 overloads)	922
5.150.13.2	ip::address::operator= (2 of 3 overloads)	922
5.150.13.3	ip::address::operator= (3 of 3 overloads)	922
5.150.14	ip::address::operator==	922
5.150.15	ip::address::operator>	923
5.150.16	ip::address::operator>=	923
5.150.17	ip::address::to_string	923
5.150.17.1	ip::address::to_string (1 of 2 overloads)	923
5.150.17.2	ip::address::to_string (2 of 2 overloads)	923
5.150.18	ip::address::to_v4	924
5.150.19	ip::address::to_v6	924
5.151	ip::address_v4	924
5.151.1	ip::address_v4::address_v4	926
5.151.1.1	ip::address_v4::address_v4 (1 of 4 overloads)	927
5.151.1.2	ip::address_v4::address_v4 (2 of 4 overloads)	927
5.151.1.3	ip::address_v4::address_v4 (3 of 4 overloads)	927
5.151.1.4	ip::address_v4::address_v4 (4 of 4 overloads)	927
5.151.2	ip::address_v4::any	927
5.151.3	ip::address_v4::broadcast	927
5.151.3.1	ip::address_v4::broadcast (1 of 2 overloads)	927
5.151.3.2	ip::address_v4::broadcast (2 of 2 overloads)	928
5.151.4	ip::address_v4::bytes_type	928
5.151.5	ip::address_v4::from_string	928
5.151.5.1	ip::address_v4::from_string (1 of 4 overloads)	928
5.151.5.2	ip::address_v4::from_string (2 of 4 overloads)	929
5.151.5.3	ip::address_v4::from_string (3 of 4 overloads)	929
5.151.5.4	ip::address_v4::from_string (4 of 4 overloads)	929
5.151.6	ip::address_v4::is_class_a	929
5.151.7	ip::address_v4::is_class_b	929
5.151.8	ip::address_v4::is_class_c	929

5.151.9	ip::address_v4::is_loopback	929
5.151.10	ip::address_v4::is_multicast	929
5.151.11	ip::address_v4::is_unspecified	930
5.151.12	ip::address_v4::loopback	930
5.151.13	ip::address_v4::make_address_v4	930
5.151.13.1	ip::address_v4::make_address_v4 (1 of 9 overloads)	931
5.151.13.2	ip::address_v4::make_address_v4 (2 of 9 overloads)	931
5.151.13.3	ip::address_v4::make_address_v4 (3 of 9 overloads)	931
5.151.13.4	ip::address_v4::make_address_v4 (4 of 9 overloads)	931
5.151.13.5	ip::address_v4::make_address_v4 (5 of 9 overloads)	931
5.151.13.6	ip::address_v4::make_address_v4 (6 of 9 overloads)	931
5.151.13.7	ip::address_v4::make_address_v4 (7 of 9 overloads)	931
5.151.13.8	ip::address_v4::make_address_v4 (8 of 9 overloads)	931
5.151.13.9	ip::address_v4::make_address_v4 (9 of 9 overloads)	932
5.151.14	ip::address_v4::make_network_v4	932
5.151.14.1	ip::address_v4::make_network_v4 (1 of 2 overloads)	932
5.151.14.2	ip::address_v4::make_network_v4 (2 of 2 overloads)	932
5.151.15	ip::address_v4::netmask	932
5.151.16	ip::address_v4::operator!=	932
5.151.17	ip::address_v4::operator<	933
5.151.18	ip::address_v4::operator<<	933
5.151.18.1	ip::address_v4::operator<< (1 of 2 overloads)	933
5.151.18.2	ip::address_v4::operator<< (2 of 2 overloads)	934
5.151.19	ip::address_v4::operator<=	934
5.151.20	ip::address_v4::operator=	934
5.151.21	ip::address_v4::operator==	935
5.151.22	ip::address_v4::operator>	935
5.151.23	ip::address_v4::operator>=	935
5.151.24	ip::address_v4::to_bytes	935
5.151.25	ip::address_v4::to_string	936
5.151.25.1	ip::address_v4::to_string (1 of 2 overloads)	936
5.151.25.2	ip::address_v4::to_string (2 of 2 overloads)	936
5.151.26	ip::address_v4::to_uint	936
5.151.27	ip::address_v4::to_ulong	936
5.151.28	ip::address_v4::uint_type	936
5.152	ip::address_v4_iterator	936
5.153	ip::address_v4_range	938
5.154	ip::address_v6	939
5.154.1	ip::address_v6::address_v6	941

5.154.1.1	ip::address_v6::address_v6 (1 of 3 overloads)	941
5.154.1.2	ip::address_v6::address_v6 (2 of 3 overloads)	941
5.154.1.3	ip::address_v6::address_v6 (3 of 3 overloads)	941
5.154.2	ip::address_v6::any	942
5.154.3	ip::address_v6::bytes_type	942
5.154.4	ip::address_v6::from_string	942
5.154.4.1	ip::address_v6::from_string (1 of 4 overloads)	942
5.154.4.2	ip::address_v6::from_string (2 of 4 overloads)	942
5.154.4.3	ip::address_v6::from_string (3 of 4 overloads)	943
5.154.4.4	ip::address_v6::from_string (4 of 4 overloads)	943
5.154.5	ip::address_v6::is_link_local	943
5.154.6	ip::address_v6::is_loopback	943
5.154.7	ip::address_v6::is_multicast	943
5.154.8	ip::address_v6::is_multicast_global	943
5.154.9	ip::address_v6::is_multicast_link_local	943
5.154.10	ip::address_v6::is_multicast_node_local	943
5.154.11	ip::address_v6::is_multicast_org_local	944
5.154.12	ip::address_v6::is_multicast_site_local	944
5.154.13	ip::address_v6::is_site_local	944
5.154.14	ip::address_v6::is_unspecified	944
5.154.15	ip::address_v6::is_v4_compatible	944
5.154.16	ip::address_v6::is_v4_mapped	944
5.154.17	ip::address_v6::loopback	944
5.154.18	ip::address_v6::make_address_v6	945
5.154.18.1	ip::address_v6::make_address_v6 (1 of 8 overloads)	945
5.154.18.2	ip::address_v6::make_address_v6 (2 of 8 overloads)	945
5.154.18.3	ip::address_v6::make_address_v6 (3 of 8 overloads)	946
5.154.18.4	ip::address_v6::make_address_v6 (4 of 8 overloads)	946
5.154.18.5	ip::address_v6::make_address_v6 (5 of 8 overloads)	946
5.154.18.6	ip::address_v6::make_address_v6 (6 of 8 overloads)	946
5.154.18.7	ip::address_v6::make_address_v6 (7 of 8 overloads)	946
5.154.18.8	ip::address_v6::make_address_v6 (8 of 8 overloads)	946
5.154.19	ip::address_v6::make_network_v6	946
5.154.20	ip::address_v6::operator!=	947
5.154.21	ip::address_v6::operator<	947
5.154.22	ip::address_v6::operator<<	947
5.154.22.1	ip::address_v6::operator<< (1 of 2 overloads)	948
5.154.22.2	ip::address_v6::operator<< (2 of 2 overloads)	948
5.154.23	ip::address_v6::operator<=	948

5.154.24	ip::address_v6::operator=	949
5.154.25	ip::address_v6::operator==	949
5.154.26	ip::address_v6::operator>	949
5.154.27	ip::address_v6::operator>=	949
5.154.28	ip::address_v6::scope_id	950
5.154.28.1	ip::address_v6::scope_id (1 of 2 overloads)	950
5.154.28.2	ip::address_v6::scope_id (2 of 2 overloads)	950
5.154.29	ip::address_v6::to_bytes	950
5.154.30	ip::address_v6::to_string	950
5.154.30.1	ip::address_v6::to_string (1 of 2 overloads)	950
5.154.30.2	ip::address_v6::to_string (2 of 2 overloads)	951
5.154.31	ip::address_v6::to_v4	951
5.154.32	ip::address_v6::v4_compatible	951
5.154.33	ip::address_v6::v4_mapped	951
5.155	ip::address_v6_iterator	951
5.156	ip::address_v6_range	952
5.157	ip::bad_address_cast	953
5.157.1	ip::bad_address_cast::bad_address_cast	954
5.157.2	ip::bad_address_cast::what	954
5.157.3	ip::bad_address_cast::~bad_address_cast	954
5.158	ip::basic_address_iterator< address_v4 >	954
5.158.1	ip::basic_address_iterator< address_v4 >::basic_address_iterator	955
5.158.1.1	ip::basic_address_iterator< address_v4 >::basic_address_iterator (1 of 2 overloads)	955
5.158.1.2	ip::basic_address_iterator< address_v4 >::basic_address_iterator (2 of 2 overloads)	956
5.158.2	ip::basic_address_iterator< address_v4 >::difference_type	956
5.158.3	ip::basic_address_iterator< address_v4 >::iterator_category	956
5.158.4	ip::basic_address_iterator< address_v4 >::operator *	956
5.158.5	ip::basic_address_iterator< address_v4 >::operator!=	956
5.158.6	ip::basic_address_iterator< address_v4 >::operator++	957
5.158.6.1	ip::basic_address_iterator< address_v4 >::operator++ (1 of 2 overloads)	957
5.158.6.2	ip::basic_address_iterator< address_v4 >::operator++ (2 of 2 overloads)	957
5.158.7	ip::basic_address_iterator< address_v4 >::operator--	957
5.158.7.1	ip::basic_address_iterator< address_v4 >::operator-- (1 of 2 overloads)	957
5.158.7.2	ip::basic_address_iterator< address_v4 >::operator-- (2 of 2 overloads)	957
5.158.8	ip::basic_address_iterator< address_v4 >::operator->	957
5.158.9	ip::basic_address_iterator< address_v4 >::operator=	958
5.158.10	ip::basic_address_iterator< address_v4 >::operator==	958
5.158.11	ip::basic_address_iterator< address_v4 >::pointer	958
5.158.12	ip::basic_address_iterator< address_v4 >::reference	958

5.158.13	ip::basic_address_iterator< address_v4 >::value_type	960
5.159	ip::basic_address_iterator< address_v6 >	963
5.159.1	ip::basic_address_iterator< address_v6 >::basic_address_iterator	964
5.159.1.1	ip::basic_address_iterator< address_v6 >::basic_address_iterator (1 of 2 overloads)	964
5.159.1.2	ip::basic_address_iterator< address_v6 >::basic_address_iterator (2 of 2 overloads)	964
5.159.2	ip::basic_address_iterator< address_v6 >::difference_type	964
5.159.3	ip::basic_address_iterator< address_v6 >::iterator_category	965
5.159.4	ip::basic_address_iterator< address_v6 >::operator *	965
5.159.5	ip::basic_address_iterator< address_v6 >::operator!=	965
5.159.6	ip::basic_address_iterator< address_v6 >::operator++	965
5.159.6.1	ip::basic_address_iterator< address_v6 >::operator++ (1 of 2 overloads)	965
5.159.6.2	ip::basic_address_iterator< address_v6 >::operator++ (2 of 2 overloads)	966
5.159.7	ip::basic_address_iterator< address_v6 >::operator--	966
5.159.7.1	ip::basic_address_iterator< address_v6 >::operator-- (1 of 2 overloads)	966
5.159.7.2	ip::basic_address_iterator< address_v6 >::operator-- (2 of 2 overloads)	966
5.159.8	ip::basic_address_iterator< address_v6 >::operator->	966
5.159.9	ip::basic_address_iterator< address_v6 >::operator=	966
5.159.10	ip::basic_address_iterator< address_v6 >::operator==	966
5.159.11	ip::basic_address_iterator< address_v6 >::pointer	967
5.159.12	ip::basic_address_iterator< address_v6 >::reference	967
5.159.13	ip::basic_address_iterator< address_v6 >::value_type	969
5.160	ip::basic_address_range< address_v4 >	972
5.160.1	ip::basic_address_range< address_v4 >::basic_address_range	972
5.160.1.1	ip::basic_address_range< address_v4 >::basic_address_range (1 of 3 overloads)	973
5.160.1.2	ip::basic_address_range< address_v4 >::basic_address_range (2 of 3 overloads)	973
5.160.1.3	ip::basic_address_range< address_v4 >::basic_address_range (3 of 3 overloads)	973
5.160.2	ip::basic_address_range< address_v4 >::begin	973
5.160.3	ip::basic_address_range< address_v4 >::empty	973
5.160.4	ip::basic_address_range< address_v4 >::end	973
5.160.5	ip::basic_address_range< address_v4 >::find	974
5.160.6	ip::basic_address_range< address_v4 >::iterator	974
5.160.7	ip::basic_address_range< address_v4 >::operator=	975
5.160.8	ip::basic_address_range< address_v4 >::size	975
5.161	ip::basic_address_range< address_v6 >	975
5.161.1	ip::basic_address_range< address_v6 >::basic_address_range	976
5.161.1.1	ip::basic_address_range< address_v6 >::basic_address_range (1 of 3 overloads)	976
5.161.1.2	ip::basic_address_range< address_v6 >::basic_address_range (2 of 3 overloads)	977
5.161.1.3	ip::basic_address_range< address_v6 >::basic_address_range (3 of 3 overloads)	977
5.161.2	ip::basic_address_range< address_v6 >::begin	977

5.161.3	ip::basic_address_range< address_v6 >::empty	977
5.161.4	ip::basic_address_range< address_v6 >::end	977
5.161.5	ip::basic_address_range< address_v6 >::find	977
5.161.6	ip::basic_address_range< address_v6 >::iterator	977
5.161.7	ip::basic_address_range< address_v6 >::operator=	979
5.162	ip::basic_endpoint	979
5.162.1	ip::basic_endpoint::address	981
5.162.1.1	ip::basic_endpoint::address (1 of 2 overloads)	981
5.162.1.2	ip::basic_endpoint::address (2 of 2 overloads)	981
5.162.2	ip::basic_endpoint::basic_endpoint	981
5.162.2.1	ip::basic_endpoint::basic_endpoint (1 of 5 overloads)	982
5.162.2.2	ip::basic_endpoint::basic_endpoint (2 of 5 overloads)	982
5.162.2.3	ip::basic_endpoint::basic_endpoint (3 of 5 overloads)	982
5.162.2.4	ip::basic_endpoint::basic_endpoint (4 of 5 overloads)	982
5.162.2.5	ip::basic_endpoint::basic_endpoint (5 of 5 overloads)	982
5.162.3	ip::basic_endpoint::capacity	982
5.162.4	ip::basic_endpoint::data	983
5.162.4.1	ip::basic_endpoint::data (1 of 2 overloads)	983
5.162.4.2	ip::basic_endpoint::data (2 of 2 overloads)	983
5.162.5	ip::basic_endpoint::data_type	983
5.162.6	ip::basic_endpoint::operator!=	983
5.162.7	ip::basic_endpoint::operator<	983
5.162.8	ip::basic_endpoint::operator<<	984
5.162.9	ip::basic_endpoint::operator<=	984
5.162.10	ip::basic_endpoint::operator=	984
5.162.10.1	ip::basic_endpoint::operator= (1 of 2 overloads)	985
5.162.10.2	ip::basic_endpoint::operator= (2 of 2 overloads)	985
5.162.11	ip::basic_endpoint::operator==	985
5.162.12	ip::basic_endpoint::operator>	985
5.162.13	ip::basic_endpoint::operator>=	985
5.162.14	ip::basic_endpoint::port	986
5.162.14.1	ip::basic_endpoint::port (1 of 2 overloads)	986
5.162.14.2	ip::basic_endpoint::port (2 of 2 overloads)	986
5.162.15	ip::basic_endpoint::protocol	986
5.162.16	ip::basic_endpoint::protocol_type	986
5.162.17	ip::basic_endpoint::resize	986
5.162.18	ip::basic_endpoint::size	987
5.163	ip::basic_resolver	987
5.163.1	ip::basic_resolver::address_configured	989

5.163.2	ip::basic_resolver::all_matching	989
5.163.3	ip::basic_resolver::async_resolve	989
5.163.3.1	ip::basic_resolver::async_resolve (1 of 6 overloads)	990
5.163.3.2	ip::basic_resolver::async_resolve (2 of 6 overloads)	990
5.163.3.3	ip::basic_resolver::async_resolve (3 of 6 overloads)	991
5.163.3.4	ip::basic_resolver::async_resolve (4 of 6 overloads)	992
5.163.3.5	ip::basic_resolver::async_resolve (5 of 6 overloads)	993
5.163.3.6	ip::basic_resolver::async_resolve (6 of 6 overloads)	994
5.163.4	ip::basic_resolver::basic_resolver	994
5.163.4.1	ip::basic_resolver::basic_resolver (1 of 2 overloads)	994
5.163.4.2	ip::basic_resolver::basic_resolver (2 of 2 overloads)	995
5.163.5	ip::basic_resolver::cancel	995
5.163.6	ip::basic_resolver::canonical_name	995
5.163.7	ip::basic_resolver::endpoint_type	995
5.163.8	ip::basic_resolver::executor_type	995
5.163.9	ip::basic_resolver::flags	996
5.163.10	ip::basic_resolver::get_executor	996
5.163.11	ip::basic_resolver::get_io_context	997
5.163.12	ip::basic_resolver::get_io_service	997
5.163.13	ip::basic_resolver::iterator	997
5.163.14	ip::basic_resolver::numeric_host	999
5.163.15	ip::basic_resolver::numeric_service	999
5.163.16	ip::basic_resolver::operator=	999
5.163.17	ip::basic_resolver::passive	999
5.163.18	ip::basic_resolver::protocol_type	999
5.163.19	ip::basic_resolver::query	1000
5.163.20	ip::basic_resolver::resolve	1001
5.163.20.1	ip::basic_resolver::resolve (1 of 12 overloads)	1002
5.163.20.2	ip::basic_resolver::resolve (2 of 12 overloads)	1003
5.163.20.3	ip::basic_resolver::resolve (3 of 12 overloads)	1003
5.163.20.4	ip::basic_resolver::resolve (4 of 12 overloads)	1004
5.163.20.5	ip::basic_resolver::resolve (5 of 12 overloads)	1005
5.163.20.6	ip::basic_resolver::resolve (6 of 12 overloads)	1005
5.163.20.7	ip::basic_resolver::resolve (7 of 12 overloads)	1006
5.163.20.8	ip::basic_resolver::resolve (8 of 12 overloads)	1007
5.163.20.9	ip::basic_resolver::resolve (9 of 12 overloads)	1008
5.163.20.10	ip::basic_resolver::resolve (10 of 12 overloads)	1008
5.163.20.11	ip::basic_resolver::resolve (11 of 12 overloads)	1009
5.163.20.12	ip::basic_resolver::resolve (12 of 12 overloads)	1010

5.163.21	ip::basic_resolver::results_type	1010
5.163.22	ip::basic_resolver::v4_mapped	1012
5.163.23	ip::basic_resolver::~basic_resolver	1012
5.164	ip::basic_resolver_entry	1012
5.164.1	ip::basic_resolver_entry::basic_resolver_entry	1013
5.164.1.1	ip::basic_resolver_entry::basic_resolver_entry (1 of 2 overloads)	1014
5.164.1.2	ip::basic_resolver_entry::basic_resolver_entry (2 of 2 overloads)	1014
5.164.2	ip::basic_resolver_entry::endpoint	1014
5.164.3	ip::basic_resolver_entry::endpoint_type	1014
5.164.4	ip::basic_resolver_entry::host_name	1014
5.164.4.1	ip::basic_resolver_entry::host_name (1 of 2 overloads)	1014
5.164.4.2	ip::basic_resolver_entry::host_name (2 of 2 overloads)	1015
5.164.5	ip::basic_resolver_entry::operator endpoint_type	1015
5.164.6	ip::basic_resolver_entry::protocol_type	1015
5.164.7	ip::basic_resolver_entry::service_name	1015
5.164.7.1	ip::basic_resolver_entry::service_name (1 of 2 overloads)	1015
5.164.7.2	ip::basic_resolver_entry::service_name (2 of 2 overloads)	1015
5.165	ip::basic_resolver_iterator	1016
5.165.1	ip::basic_resolver_iterator::basic_resolver_iterator	1017
5.165.1.1	ip::basic_resolver_iterator::basic_resolver_iterator (1 of 3 overloads)	1017
5.165.1.2	ip::basic_resolver_iterator::basic_resolver_iterator (2 of 3 overloads)	1018
5.165.1.3	ip::basic_resolver_iterator::basic_resolver_iterator (3 of 3 overloads)	1018
5.165.2	ip::basic_resolver_iterator::dereference	1018
5.165.3	ip::basic_resolver_iterator::difference_type	1018
5.165.4	ip::basic_resolver_iterator::equal	1018
5.165.5	ip::basic_resolver_iterator::increment	1018
5.165.6	ip::basic_resolver_iterator::index_	1018
5.165.7	ip::basic_resolver_iterator::iterator_category	1018
5.165.8	ip::basic_resolver_iterator::operator *	1019
5.165.9	ip::basic_resolver_iterator::operator!=	1019
5.165.10	ip::basic_resolver_iterator::operator++	1019
5.165.10.1	ip::basic_resolver_iterator::operator++ (1 of 2 overloads)	1019
5.165.10.2	ip::basic_resolver_iterator::operator++ (2 of 2 overloads)	1019
5.165.11	ip::basic_resolver_iterator::operator->	1020
5.165.12	ip::basic_resolver_iterator::operator=	1020
5.165.12.1	ip::basic_resolver_iterator::operator= (1 of 2 overloads)	1020
5.165.12.2	ip::basic_resolver_iterator::operator= (2 of 2 overloads)	1020
5.165.13	ip::basic_resolver_iterator::operator==	1020
5.165.14	ip::basic_resolver_iterator::pointer	1020

5.165.15	ip::basic_resolver_iterator::reference	1021
5.165.16	ip::basic_resolver_iterator::value_type	1022
5.165.17	ip::basic_resolver_iterator::values_	1022
5.166	ip::basic_resolver_query	1023
5.166.1	ip::basic_resolver_query::address_configured	1024
5.166.2	ip::basic_resolver_query::all_matching	1024
5.166.3	ip::basic_resolver_query::basic_resolver_query	1024
5.166.3.1	ip::basic_resolver_query::basic_resolver_query (1 of 4 overloads)	1025
5.166.3.2	ip::basic_resolver_query::basic_resolver_query (2 of 4 overloads)	1025
5.166.3.3	ip::basic_resolver_query::basic_resolver_query (3 of 4 overloads)	1026
5.166.3.4	ip::basic_resolver_query::basic_resolver_query (4 of 4 overloads)	1026
5.166.4	ip::basic_resolver_query::canonical_name	1027
5.166.5	ip::basic_resolver_query::flags	1027
5.166.6	ip::basic_resolver_query::hints	1027
5.166.7	ip::basic_resolver_query::host_name	1028
5.166.8	ip::basic_resolver_query::numeric_host	1028
5.166.9	ip::basic_resolver_query::numeric_service	1028
5.166.10	ip::basic_resolver_query::passive	1028
5.166.11	ip::basic_resolver_query::protocol_type	1028
5.166.12	ip::basic_resolver_query::service_name	1028
5.166.13	ip::basic_resolver_query::v4_mapped	1028
5.167	ip::basic_resolver_results	1029
5.167.1	ip::basic_resolver_results::basic_resolver_results	1031
5.167.1.1	ip::basic_resolver_results::basic_resolver_results (1 of 3 overloads)	1031
5.167.1.2	ip::basic_resolver_results::basic_resolver_results (2 of 3 overloads)	1031
5.167.1.3	ip::basic_resolver_results::basic_resolver_results (3 of 3 overloads)	1031
5.167.2	ip::basic_resolver_results::begin	1031
5.167.3	ip::basic_resolver_results::cbegin	1032
5.167.4	ip::basic_resolver_results::cend	1032
5.167.5	ip::basic_resolver_results::const_iterator	1032
5.167.6	ip::basic_resolver_results::const_reference	1033
5.167.7	ip::basic_resolver_results::dereference	1034
5.167.8	ip::basic_resolver_results::difference_type	1034
5.167.9	ip::basic_resolver_results::empty	1035
5.167.10	ip::basic_resolver_results::end	1035
5.167.11	ip::basic_resolver_results::endpoint_type	1035
5.167.12	ip::basic_resolver_results::equal	1035
5.167.13	ip::basic_resolver_results::increment	1035
5.167.14	ip::basic_resolver_results::index_	1035

5.167.15	ip::basic_resolver_results::iterator	1035
5.167.16	ip::basic_resolver_results::iterator_category	1037
5.167.17	ip::basic_resolver_results::max_size	1037
5.167.18	ip::basic_resolver_results::operator *	1037
5.167.19	ip::basic_resolver_results::operator!=	1038
5.167.19.1	ip::basic_resolver_results::operator!= (1 of 2 overloads)	1038
5.167.19.2	ip::basic_resolver_results::operator!= (2 of 2 overloads)	1038
5.167.20	ip::basic_resolver_results::operator++	1038
5.167.20.1	ip::basic_resolver_results::operator++ (1 of 2 overloads)	1039
5.167.20.2	ip::basic_resolver_results::operator++ (2 of 2 overloads)	1039
5.167.21	ip::basic_resolver_results::operator->	1039
5.167.22	ip::basic_resolver_results::operator=	1039
5.167.22.1	ip::basic_resolver_results::operator= (1 of 2 overloads)	1039
5.167.22.2	ip::basic_resolver_results::operator= (2 of 2 overloads)	1039
5.167.23	ip::basic_resolver_results::operator==	1040
5.167.23.1	ip::basic_resolver_results::operator== (1 of 2 overloads)	1040
5.167.23.2	ip::basic_resolver_results::operator== (2 of 2 overloads)	1040
5.167.24	ip::basic_resolver_results::pointer	1040
5.167.25	ip::basic_resolver_results::protocol_type	1041
5.167.26	ip::basic_resolver_results::reference	1041
5.167.27	ip::basic_resolver_results::size	1042
5.167.28	ip::basic_resolver_results::size_type	1042
5.167.29	ip::basic_resolver_results::swap	1042
5.167.30	ip::basic_resolver_results::value_type	1042
5.167.31	ip::basic_resolver_results::values_	1043
5.168	ip::host_name	1043
5.168.1	ip::host_name (1 of 2 overloads)	1044
5.168.2	ip::host_name (2 of 2 overloads)	1044
5.169	ip::icmp	1044
5.169.1	ip::icmp::endpoint	1045
5.169.2	ip::icmp::family	1047
5.169.3	ip::icmp::operator!=	1047
5.169.4	ip::icmp::operator==	1047
5.169.5	ip::icmp::protocol	1047
5.169.6	ip::icmp::resolver	1047
5.169.7	ip::icmp::socket	1049
5.169.8	ip::icmp::type	1053
5.169.9	ip::icmp::v4	1053
5.169.10	ip::icmp::v6	1053

5.170	ip::multicast::enable_loopback	1053
5.171	ip::multicast::hops	1054
5.172	ip::multicast::join_group	1054
5.173	ip::multicast::leave_group	1055
5.174	ip::multicast::outbound_interface	1055
5.175	ip::network_v4	1056
5.175.1	ip::network_v4::address	1057
5.175.2	ip::network_v4::broadcast	1057
5.175.3	ip::network_v4::canonical	1057
5.175.4	ip::network_v4::hosts	1057
5.175.5	ip::network_v4::is_host	1057
5.175.6	ip::network_v4::is_subnet_of	1058
5.175.7	ip::network_v4::make_network_v4	1058
5.175.7.1	ip::network_v4::make_network_v4 (1 of 6 overloads)	1058
5.175.7.2	ip::network_v4::make_network_v4 (2 of 6 overloads)	1058
5.175.7.3	ip::network_v4::make_network_v4 (3 of 6 overloads)	1058
5.175.7.4	ip::network_v4::make_network_v4 (4 of 6 overloads)	1059
5.175.7.5	ip::network_v4::make_network_v4 (5 of 6 overloads)	1059
5.175.7.6	ip::network_v4::make_network_v4 (6 of 6 overloads)	1059
5.175.8	ip::network_v4::netmask	1059
5.175.9	ip::network_v4::network	1059
5.175.10	ip::network_v4::network_v4	1059
5.175.10.1	ip::network_v4::network_v4 (1 of 4 overloads)	1060
5.175.10.2	ip::network_v4::network_v4 (2 of 4 overloads)	1060
5.175.10.3	ip::network_v4::network_v4 (3 of 4 overloads)	1060
5.175.10.4	ip::network_v4::network_v4 (4 of 4 overloads)	1060
5.175.11	ip::network_v4::operator!=	1060
5.175.12	ip::network_v4::operator=	1060
5.175.13	ip::network_v4::operator==	1061
5.175.14	ip::network_v4::prefix_length	1061
5.175.15	ip::network_v4::to_string	1061
5.175.15.1	ip::network_v4::to_string (1 of 2 overloads)	1061
5.175.15.2	ip::network_v4::to_string (2 of 2 overloads)	1061
5.176	ip::network_v6	1061
5.176.1	ip::network_v6::address	1063
5.176.2	ip::network_v6::canonical	1063
5.176.3	ip::network_v6::hosts	1063
5.176.4	ip::network_v6::is_host	1063
5.176.5	ip::network_v6::is_subnet_of	1063

5.176.6	ip::network_v6::make_network_v6	1063
5.176.6.1	ip::network_v6::make_network_v6 (1 of 6 overloads)	1064
5.176.6.2	ip::network_v6::make_network_v6 (2 of 6 overloads)	1064
5.176.6.3	ip::network_v6::make_network_v6 (3 of 6 overloads)	1064
5.176.6.4	ip::network_v6::make_network_v6 (4 of 6 overloads)	1064
5.176.6.5	ip::network_v6::make_network_v6 (5 of 6 overloads)	1064
5.176.6.6	ip::network_v6::make_network_v6 (6 of 6 overloads)	1064
5.176.7	ip::network_v6::network	1065
5.176.8	ip::network_v6::network_v6	1065
5.176.8.1	ip::network_v6::network_v6 (1 of 3 overloads)	1065
5.176.8.2	ip::network_v6::network_v6 (2 of 3 overloads)	1065
5.176.8.3	ip::network_v6::network_v6 (3 of 3 overloads)	1065
5.176.9	ip::network_v6::operator!=	1065
5.176.10	ip::network_v6::operator=	1066
5.176.11	ip::network_v6::operator==	1066
5.176.12	ip::network_v6::prefix_length	1066
5.176.13	ip::network_v6::to_string	1066
5.176.13.1	ip::network_v6::to_string (1 of 2 overloads)	1066
5.176.13.2	ip::network_v6::to_string (2 of 2 overloads)	1066
5.177	ip::resolver_base	1067
5.177.1	ip::resolver_base::address_configured	1068
5.177.2	ip::resolver_base::all_matching	1068
5.177.3	ip::resolver_base::canonical_name	1068
5.177.4	ip::resolver_base::flags	1068
5.177.5	ip::resolver_base::numeric_host	1068
5.177.6	ip::resolver_base::numeric_service	1068
5.177.7	ip::resolver_base::passive	1068
5.177.8	ip::resolver_base::v4_mapped	1069
5.177.9	ip::resolver_base::~resolver_base	1069
5.178	ip::resolver_query_base	1069
5.178.1	ip::resolver_query_base::address_configured	1070
5.178.2	ip::resolver_query_base::all_matching	1070
5.178.3	ip::resolver_query_base::canonical_name	1070
5.178.4	ip::resolver_query_base::flags	1070
5.178.5	ip::resolver_query_base::numeric_host	1071
5.178.6	ip::resolver_query_base::numeric_service	1071
5.178.7	ip::resolver_query_base::passive	1071
5.178.8	ip::resolver_query_base::v4_mapped	1071
5.178.9	ip::resolver_query_base::~resolver_query_base	1071

5.179	ip::tcp	1071
5.179.1	ip::tcp::acceptor	1072
5.179.2	ip::tcp::endpoint	1076
5.179.3	ip::tcp::family	1077
5.179.4	ip::tcp::iostream	1078
5.179.5	ip::tcp::no_delay	1079
5.179.6	ip::tcp::operator!=	1079
5.179.7	ip::tcp::operator==	1080
5.179.8	ip::tcp::protocol	1080
5.179.9	ip::tcp::resolver	1080
5.179.10	ip::tcp::socket	1082
5.179.11	ip::tcp::type	1086
5.179.12	ip::tcp::v4	1086
5.179.13	ip::tcp::v6	1086
5.180	ip::udp	1086
5.180.1	ip::udp::endpoint	1087
5.180.2	ip::udp::family	1089
5.180.3	ip::udp::operator!=	1089
5.180.4	ip::udp::operator==	1089
5.180.5	ip::udp::protocol	1090
5.180.6	ip::udp::resolver	1090
5.180.7	ip::udp::socket	1092
5.180.8	ip::udp::type	1095
5.180.9	ip::udp::v4	1095
5.180.10	ip::udp::v6	1095
5.181	ip::unicast::hops	1095
5.182	ip::v4_mapped_t	1096
5.183	ip::v6_only	1096
5.184	is_const_buffer_sequence	1097
5.185	is_dynamic_buffer	1097
5.186	is_endpoint_sequence	1097
5.186.1	is_endpoint_sequence::value	1098
5.187	is_executor	1098
5.188	is_match_condition	1098
5.188.1	is_match_condition::value	1099
5.189	is_mutable_buffer_sequence	1099
5.190	is_read_buffered	1099
5.190.1	is_read_buffered::value	1099
5.191	is_write_buffered	1100

5.191.1	is_write_buffered::value	1100
5.192	local::basic_endpoint	1100
5.192.1	local::basic_endpoint::basic_endpoint	1102
5.192.1.1	local::basic_endpoint::basic_endpoint (1 of 4 overloads)	1102
5.192.1.2	local::basic_endpoint::basic_endpoint (2 of 4 overloads)	1102
5.192.1.3	local::basic_endpoint::basic_endpoint (3 of 4 overloads)	1102
5.192.1.4	local::basic_endpoint::basic_endpoint (4 of 4 overloads)	1102
5.192.2	local::basic_endpoint::capacity	1103
5.192.3	local::basic_endpoint::data	1103
5.192.3.1	local::basic_endpoint::data (1 of 2 overloads)	1103
5.192.3.2	local::basic_endpoint::data (2 of 2 overloads)	1103
5.192.4	local::basic_endpoint::data_type	1103
5.192.5	local::basic_endpoint::operator!=	1103
5.192.6	local::basic_endpoint::operator<	1104
5.192.7	local::basic_endpoint::operator<<	1104
5.192.8	local::basic_endpoint::operator<=	1104
5.192.9	local::basic_endpoint::operator=	1104
5.192.10	local::basic_endpoint::operator==	1105
5.192.11	local::basic_endpoint::operator>	1105
5.192.12	local::basic_endpoint::operator>=	1105
5.192.13	local::basic_endpoint::path	1105
5.192.13.1	local::basic_endpoint::path (1 of 3 overloads)	1106
5.192.13.2	local::basic_endpoint::path (2 of 3 overloads)	1106
5.192.13.3	local::basic_endpoint::path (3 of 3 overloads)	1106
5.192.14	local::basic_endpoint::protocol	1106
5.192.15	local::basic_endpoint::protocol_type	1106
5.192.16	local::basic_endpoint::resize	1106
5.192.17	local::basic_endpoint::size	1106
5.193	local::connect_pair	1107
5.193.1	local::connect_pair (1 of 2 overloads)	1107
5.193.2	local::connect_pair (2 of 2 overloads)	1107
5.194	local::datagram_protocol	1107
5.194.1	local::datagram_protocol::endpoint	1108
5.194.2	local::datagram_protocol::family	1110
5.194.3	local::datagram_protocol::protocol	1110
5.194.4	local::datagram_protocol::socket	1110
5.194.5	local::datagram_protocol::type	1114
5.195	local::stream_protocol	1114
5.195.1	local::stream_protocol::acceptor	1115

5.195.2	local::stream_protocol::endpoint	1118
5.195.3	local::stream_protocol::family	1119
5.195.4	local::stream_protocol::iostream	1119
5.195.5	local::stream_protocol::protocol	1121
5.195.6	local::stream_protocol::socket	1121
5.195.7	local::stream_protocol::type	1124
5.196	make_work_guard	1124
5.196.1	make_work_guard (1 of 5 overloads)	1125
5.196.2	make_work_guard (2 of 5 overloads)	1126
5.196.3	make_work_guard (3 of 5 overloads)	1126
5.196.4	make_work_guard (4 of 5 overloads)	1126
5.196.5	make_work_guard (5 of 5 overloads)	1126
5.197	mutable_buffer	1126
5.197.1	mutable_buffer::data	1127
5.197.2	mutable_buffer::mutable_buffer	1127
5.197.2.1	mutable_buffer::mutable_buffer (1 of 2 overloads)	1128
5.197.2.2	mutable_buffer::mutable_buffer (2 of 2 overloads)	1128
5.197.3	mutable_buffer::operator+	1128
5.197.3.1	mutable_buffer::operator+ (1 of 2 overloads)	1128
5.197.3.2	mutable_buffer::operator+ (2 of 2 overloads)	1128
5.197.4	mutable_buffer::operator+=	1128
5.197.5	mutable_buffer::size	1129
5.198	mutable_buffers_1	1129
5.198.1	mutable_buffers_1::begin	1130
5.198.2	mutable_buffers_1::const_iterator	1130
5.198.3	mutable_buffers_1::data	1130
5.198.4	mutable_buffers_1::end	1130
5.198.5	mutable_buffers_1::mutable_buffers_1	1130
5.198.5.1	mutable_buffers_1::mutable_buffers_1 (1 of 2 overloads)	1131
5.198.5.2	mutable_buffers_1::mutable_buffers_1 (2 of 2 overloads)	1131
5.198.6	mutable_buffers_1::operator+	1131
5.198.6.1	mutable_buffers_1::operator+ (1 of 2 overloads)	1131
5.198.6.2	mutable_buffers_1::operator+ (2 of 2 overloads)	1131
5.198.7	mutable_buffers_1::operator+=	1131
5.198.8	mutable_buffers_1::size	1132
5.198.9	mutable_buffers_1::value_type	1132
5.199	null_buffers	1133
5.199.1	null_buffers::begin	1133
5.199.2	null_buffers::const_iterator	1133

5.199.3	null_buffers::end	1134
5.199.4	null_buffers::value_type	1134
5.200	operator<<	1135
5.201	placeholders::bytes_transferred	1135
5.202	placeholders::endpoint	1135
5.203	placeholders::error	1135
5.204	placeholders::iterator	1136
5.205	placeholders::results	1136
5.206	placeholders::signal_number	1136
5.207	posix::descriptor	1136
5.207.1	posix::descriptor::assign	1138
5.207.1.1	posix::descriptor::assign (1 of 2 overloads)	1138
5.207.1.2	posix::descriptor::assign (2 of 2 overloads)	1139
5.207.2	posix::descriptor::async_wait	1139
5.207.3	posix::descriptor::bytes_readable	1139
5.207.4	posix::descriptor::cancel	1140
5.207.4.1	posix::descriptor::cancel (1 of 2 overloads)	1140
5.207.4.2	posix::descriptor::cancel (2 of 2 overloads)	1140
5.207.5	posix::descriptor::close	1141
5.207.5.1	posix::descriptor::close (1 of 2 overloads)	1141
5.207.5.2	posix::descriptor::close (2 of 2 overloads)	1141
5.207.6	posix::descriptor::descriptor	1141
5.207.6.1	posix::descriptor::descriptor (1 of 3 overloads)	1142
5.207.6.2	posix::descriptor::descriptor (2 of 3 overloads)	1142
5.207.6.3	posix::descriptor::descriptor (3 of 3 overloads)	1142
5.207.7	posix::descriptor::executor_type	1143
5.207.8	posix::descriptor::get_executor	1143
5.207.9	posix::descriptor::get_io_context	1143
5.207.10	posix::descriptor::get_io_service	1144
5.207.11	posix::descriptor::io_control	1144
5.207.11.1	posix::descriptor::io_control (1 of 2 overloads)	1144
5.207.11.2	posix::descriptor::io_control (2 of 2 overloads)	1145
5.207.12	posix::descriptor::is_open	1145
5.207.13	posix::descriptor::lowest_layer	1146
5.207.13.1	posix::descriptor::lowest_layer (1 of 2 overloads)	1146
5.207.13.2	posix::descriptor::lowest_layer (2 of 2 overloads)	1146
5.207.14	posix::descriptor::lowest_layer_type	1146
5.207.15	posix::descriptor::native_handle	1148
5.207.16	posix::descriptor::native_handle_type	1148

5.207.17	posix::descriptor::native_non_blocking	1148
5.207.17.1	posix::descriptor::native_non_blocking (1 of 3 overloads)	1149
5.207.17.2	posix::descriptor::native_non_blocking (2 of 3 overloads)	1149
5.207.17.3	posix::descriptor::native_non_blocking (3 of 3 overloads)	1149
5.207.18	posix::descriptor::non_blocking	1150
5.207.18.1	posix::descriptor::non_blocking (1 of 3 overloads)	1150
5.207.18.2	posix::descriptor::non_blocking (2 of 3 overloads)	1150
5.207.18.3	posix::descriptor::non_blocking (3 of 3 overloads)	1151
5.207.19	posix::descriptor::operator=	1151
5.207.20	posix::descriptor::release	1151
5.207.21	posix::descriptor::wait	1152
5.207.21.1	posix::descriptor::wait (1 of 2 overloads)	1152
5.207.21.2	posix::descriptor::wait (2 of 2 overloads)	1152
5.207.22	posix::descriptor::wait_type	1153
5.207.23	posix::descriptor::~descriptor	1153
5.208	posix::descriptor_base	1153
5.208.1	posix::descriptor_base::bytes_readable	1154
5.208.2	posix::descriptor_base::wait_type	1154
5.208.3	posix::descriptor_base::~descriptor_base	1154
5.209	posix::stream_descriptor	1154
5.209.1	posix::stream_descriptor::assign	1156
5.209.1.1	posix::stream_descriptor::assign (1 of 2 overloads)	1156
5.209.1.2	posix::stream_descriptor::assign (2 of 2 overloads)	1157
5.209.2	posix::stream_descriptor::async_read_some	1157
5.209.3	posix::stream_descriptor::async_wait	1158
5.209.4	posix::stream_descriptor::async_write_some	1158
5.209.5	posix::stream_descriptor::bytes_readable	1159
5.209.6	posix::stream_descriptor::cancel	1160
5.209.6.1	posix::stream_descriptor::cancel (1 of 2 overloads)	1160
5.209.6.2	posix::stream_descriptor::cancel (2 of 2 overloads)	1160
5.209.7	posix::stream_descriptor::close	1160
5.209.7.1	posix::stream_descriptor::close (1 of 2 overloads)	1161
5.209.7.2	posix::stream_descriptor::close (2 of 2 overloads)	1161
5.209.8	posix::stream_descriptor::executor_type	1161
5.209.9	posix::stream_descriptor::get_executor	1162
5.209.10	posix::stream_descriptor::get_io_context	1162
5.209.11	posix::stream_descriptor::get_io_service	1162
5.209.12	posix::stream_descriptor::io_control	1163
5.209.12.1	posix::stream_descriptor::io_control (1 of 2 overloads)	1163

5.209.12.2	<code>posix::stream_descriptor::io_control</code> (2 of 2 overloads)	1163
5.209.13	<code>posix::stream_descriptor::is_open</code>	1164
5.209.14	<code>posix::stream_descriptor::lowest_layer</code>	1164
5.209.14.1	<code>posix::stream_descriptor::lowest_layer</code> (1 of 2 overloads)	1164
5.209.14.2	<code>posix::stream_descriptor::lowest_layer</code> (2 of 2 overloads)	1165
5.209.15	<code>posix::stream_descriptor::lowest_layer_type</code>	1165
5.209.16	<code>posix::stream_descriptor::native_handle</code>	1167
5.209.17	<code>posix::stream_descriptor::native_handle_type</code>	1167
5.209.18	<code>posix::stream_descriptor::native_non_blocking</code>	1167
5.209.18.1	<code>posix::stream_descriptor::native_non_blocking</code> (1 of 3 overloads)	1167
5.209.18.2	<code>posix::stream_descriptor::native_non_blocking</code> (2 of 3 overloads)	1168
5.209.18.3	<code>posix::stream_descriptor::native_non_blocking</code> (3 of 3 overloads)	1168
5.209.19	<code>posix::stream_descriptor::non_blocking</code>	1169
5.209.19.1	<code>posix::stream_descriptor::non_blocking</code> (1 of 3 overloads)	1169
5.209.19.2	<code>posix::stream_descriptor::non_blocking</code> (2 of 3 overloads)	1169
5.209.19.3	<code>posix::stream_descriptor::non_blocking</code> (3 of 3 overloads)	1170
5.209.20	<code>posix::stream_descriptor::operator=</code>	1170
5.209.21	<code>posix::stream_descriptor::read_some</code>	1171
5.209.21.1	<code>posix::stream_descriptor::read_some</code> (1 of 2 overloads)	1171
5.209.21.2	<code>posix::stream_descriptor::read_some</code> (2 of 2 overloads)	1172
5.209.22	<code>posix::stream_descriptor::release</code>	1172
5.209.23	<code>posix::stream_descriptor::stream_descriptor</code>	1172
5.209.23.1	<code>posix::stream_descriptor::stream_descriptor</code> (1 of 3 overloads)	1173
5.209.23.2	<code>posix::stream_descriptor::stream_descriptor</code> (2 of 3 overloads)	1173
5.209.23.3	<code>posix::stream_descriptor::stream_descriptor</code> (3 of 3 overloads)	1173
5.209.24	<code>posix::stream_descriptor::wait</code>	1174
5.209.24.1	<code>posix::stream_descriptor::wait</code> (1 of 2 overloads)	1174
5.209.24.2	<code>posix::stream_descriptor::wait</code> (2 of 2 overloads)	1174
5.209.25	<code>posix::stream_descriptor::wait_type</code>	1175
5.209.26	<code>posix::stream_descriptor::write_some</code>	1175
5.209.26.1	<code>posix::stream_descriptor::write_some</code> (1 of 2 overloads)	1175
5.209.26.2	<code>posix::stream_descriptor::write_some</code> (2 of 2 overloads)	1176
5.210	<code>post</code>	1177
5.210.1	<code>post</code> (1 of 3 overloads)	1177
5.210.2	<code>post</code> (2 of 3 overloads)	1178
5.210.3	<code>post</code> (3 of 3 overloads)	1178
5.211	<code>read</code>	1178
5.211.1	<code>read</code> (1 of 12 overloads)	1181
5.211.2	<code>read</code> (2 of 12 overloads)	1182

5.211.3	read (3 of 12 overloads)	1183
5.211.4	read (4 of 12 overloads)	1184
5.211.5	read (5 of 12 overloads)	1185
5.211.6	read (6 of 12 overloads)	1186
5.211.7	read (7 of 12 overloads)	1186
5.211.8	read (8 of 12 overloads)	1187
5.211.9	read (9 of 12 overloads)	1188
5.211.10	read (10 of 12 overloads)	1189
5.211.11	read (11 of 12 overloads)	1190
5.211.12	read (12 of 12 overloads)	1191
5.212	read_at	1191
5.212.1	read_at (1 of 8 overloads)	1193
5.212.2	read_at (2 of 8 overloads)	1194
5.212.3	read_at (3 of 8 overloads)	1195
5.212.4	read_at (4 of 8 overloads)	1196
5.212.5	read_at (5 of 8 overloads)	1197
5.212.6	read_at (6 of 8 overloads)	1198
5.212.7	read_at (7 of 8 overloads)	1199
5.212.8	read_at (8 of 8 overloads)	1200
5.213	read_until	1201
5.213.1	read_until (1 of 16 overloads)	1203
5.213.2	read_until (2 of 16 overloads)	1205
5.213.3	read_until (3 of 16 overloads)	1205
5.213.4	read_until (4 of 16 overloads)	1207
5.213.5	read_until (5 of 16 overloads)	1207
5.213.6	read_until (6 of 16 overloads)	1209
5.213.7	read_until (7 of 16 overloads)	1209
5.213.8	read_until (8 of 16 overloads)	1211
5.213.9	read_until (9 of 16 overloads)	1213
5.213.10	read_until (10 of 16 overloads)	1214
5.213.11	read_until (11 of 16 overloads)	1215
5.213.12	read_until (12 of 16 overloads)	1216
5.213.13	read_until (13 of 16 overloads)	1217
5.213.14	read_until (14 of 16 overloads)	1218
5.213.15	read_until (15 of 16 overloads)	1219
5.213.16	read_until (16 of 16 overloads)	1220
5.214	resolver_errc::try_again	1221
5.215	serial_port	1222
5.215.1	serial_port::assign	1223

5.215.1.1	serial_port::assign (1 of 2 overloads)	1223
5.215.1.2	serial_port::assign (2 of 2 overloads)	1224
5.215.2	serial_port::async_read_some	1224
5.215.3	serial_port::async_write_some	1225
5.215.4	serial_port::cancel	1225
5.215.4.1	serial_port::cancel (1 of 2 overloads)	1226
5.215.4.2	serial_port::cancel (2 of 2 overloads)	1226
5.215.5	serial_port::close	1226
5.215.5.1	serial_port::close (1 of 2 overloads)	1226
5.215.5.2	serial_port::close (2 of 2 overloads)	1227
5.215.6	serial_port::executor_type	1227
5.215.7	serial_port::get_executor	1228
5.215.8	serial_port::get_io_context	1228
5.215.9	serial_port::get_io_service	1228
5.215.10	serial_port::get_option	1228
5.215.10.1	serial_port::get_option (1 of 2 overloads)	1229
5.215.10.2	serial_port::get_option (2 of 2 overloads)	1229
5.215.11	serial_port::is_open	1229
5.215.12	serial_port::lowest_layer	1229
5.215.12.1	serial_port::lowest_layer (1 of 2 overloads)	1230
5.215.12.2	serial_port::lowest_layer (2 of 2 overloads)	1230
5.215.13	serial_port::lowest_layer_type	1230
5.215.14	serial_port::native_handle	1232
5.215.15	serial_port::native_handle_type	1232
5.215.16	serial_port::open	1232
5.215.16.1	serial_port::open (1 of 2 overloads)	1232
5.215.16.2	serial_port::open (2 of 2 overloads)	1233
5.215.17	serial_port::operator=	1233
5.215.18	serial_port::read_some	1233
5.215.18.1	serial_port::read_some (1 of 2 overloads)	1234
5.215.18.2	serial_port::read_some (2 of 2 overloads)	1234
5.215.19	serial_port::send_break	1235
5.215.19.1	serial_port::send_break (1 of 2 overloads)	1235
5.215.19.2	serial_port::send_break (2 of 2 overloads)	1235
5.215.20	serial_port::serial_port	1236
5.215.20.1	serial_port::serial_port (1 of 5 overloads)	1236
5.215.20.2	serial_port::serial_port (2 of 5 overloads)	1236
5.215.20.3	serial_port::serial_port (3 of 5 overloads)	1237
5.215.20.4	serial_port::serial_port (4 of 5 overloads)	1237

5.215.20.5	serial_port::serial_port (5 of 5 overloads)	1237
5.215.21	serial_port::set_option	1238
5.215.21.1	serial_port::set_option (1 of 2 overloads)	1238
5.215.21.2	serial_port::set_option (2 of 2 overloads)	1238
5.215.22	serial_port::write_some	1239
5.215.22.1	serial_port::write_some (1 of 2 overloads)	1239
5.215.22.2	serial_port::write_some (2 of 2 overloads)	1240
5.215.23	serial_port::~serial_port	1240
5.216	serial_port_base	1240
5.216.1	serial_port_base::~serial_port_base	1241
5.217	serial_port_base::baud_rate	1241
5.217.1	serial_port_base::baud_rate::baud_rate	1242
5.217.2	serial_port_base::baud_rate::load	1242
5.217.3	serial_port_base::baud_rate::store	1242
5.217.4	serial_port_base::baud_rate::value	1242
5.218	serial_port_base::character_size	1242
5.218.1	serial_port_base::character_size::character_size	1243
5.218.2	serial_port_base::character_size::load	1243
5.218.3	serial_port_base::character_size::store	1243
5.218.4	serial_port_base::character_size::value	1243
5.219	serial_port_base::flow_control	1243
5.219.1	serial_port_base::flow_control::flow_control	1244
5.219.2	serial_port_base::flow_control::load	1244
5.219.3	serial_port_base::flow_control::store	1244
5.219.4	serial_port_base::flow_control::type	1244
5.219.5	serial_port_base::flow_control::value	1244
5.220	serial_port_base::parity	1244
5.220.1	serial_port_base::parity::load	1245
5.220.2	serial_port_base::parity::parity	1245
5.220.3	serial_port_base::parity::store	1245
5.220.4	serial_port_base::parity::type	1245
5.220.5	serial_port_base::parity::value	1245
5.221	serial_port_base::stop_bits	1246
5.221.1	serial_port_base::stop_bits::load	1246
5.221.2	serial_port_base::stop_bits::stop_bits	1246
5.221.3	serial_port_base::stop_bits::store	1246
5.221.4	serial_port_base::stop_bits::type	1246
5.221.5	serial_port_base::stop_bits::value	1247
5.222	service_already_exists	1247

5.222.1	service_already_exists::service_already_exists	1247
5.223	signal_set	1247
5.223.1	signal_set::add	1249
5.223.1.1	signal_set::add (1 of 2 overloads)	1249
5.223.1.2	signal_set::add (2 of 2 overloads)	1250
5.223.2	signal_set::async_wait	1250
5.223.3	signal_set::cancel	1250
5.223.3.1	signal_set::cancel (1 of 2 overloads)	1251
5.223.3.2	signal_set::cancel (2 of 2 overloads)	1251
5.223.4	signal_set::clear	1252
5.223.4.1	signal_set::clear (1 of 2 overloads)	1252
5.223.4.2	signal_set::clear (2 of 2 overloads)	1252
5.223.5	signal_set::executor_type	1252
5.223.6	signal_set::get_executor	1253
5.223.7	signal_set::get_io_context	1253
5.223.8	signal_set::get_io_service	1254
5.223.9	signal_set::remove	1254
5.223.9.1	signal_set::remove (1 of 2 overloads)	1254
5.223.9.2	signal_set::remove (2 of 2 overloads)	1254
5.223.10	signal_set::signal_set	1255
5.223.10.1	signal_set::signal_set (1 of 4 overloads)	1255
5.223.10.2	signal_set::signal_set (2 of 4 overloads)	1256
5.223.10.3	signal_set::signal_set (3 of 4 overloads)	1256
5.223.10.4	signal_set::signal_set (4 of 4 overloads)	1257
5.223.11	signal_set::~signal_set	1257
5.224	socket_base	1257
5.224.1	socket_base::broadcast	1259
5.224.2	socket_base::bytes_readable	1260
5.224.3	socket_base::debug	1260
5.224.4	socket_base::do_not_route	1261
5.224.5	socket_base::enable_connection_aborted	1261
5.224.6	socket_base::keep_alive	1262
5.224.7	socket_base::linger	1262
5.224.8	socket_base::max_connections	1263
5.224.9	socket_base::max_listen_connections	1263
5.224.10	socket_base::message_do_not_route	1263
5.224.11	socket_base::message_end_of_record	1263
5.224.12	socket_base::message_flags	1263
5.224.13	socket_base::message_out_of_band	1264

5.224.14	socket_base::message_peek	1264
5.224.15	socket_base::out_of_band_inline	1264
5.224.16	socket_base::receive_buffer_size	1264
5.224.17	socket_base::receive_low_watermark	1265
5.224.18	socket_base::reuse_address	1266
5.224.19	socket_base::send_buffer_size	1266
5.224.20	socket_base::send_low_watermark	1267
5.224.21	socket_base::shutdown_type	1267
5.224.22	socket_base::wait_type	1267
5.224.23	socket_base::~socket_base	1268
5.225	spawn	1268
5.225.1	spawn (1 of 7 overloads)	1270
5.225.2	spawn (2 of 7 overloads)	1270
5.225.3	spawn (3 of 7 overloads)	1271
5.225.4	spawn (4 of 7 overloads)	1271
5.225.5	spawn (5 of 7 overloads)	1272
5.225.6	spawn (6 of 7 overloads)	1272
5.225.7	spawn (7 of 7 overloads)	1272
5.226	ssl::context	1273
5.226.1	ssl::context::add_certificate_authority	1275
5.226.1.1	ssl::context::add_certificate_authority (1 of 2 overloads)	1275
5.226.1.2	ssl::context::add_certificate_authority (2 of 2 overloads)	1276
5.226.2	ssl::context::add_verify_path	1276
5.226.2.1	ssl::context::add_verify_path (1 of 2 overloads)	1276
5.226.2.2	ssl::context::add_verify_path (2 of 2 overloads)	1277
5.226.3	ssl::context::clear_options	1277
5.226.3.1	ssl::context::clear_options (1 of 2 overloads)	1277
5.226.3.2	ssl::context::clear_options (2 of 2 overloads)	1278
5.226.4	ssl::context::context	1278
5.226.4.1	ssl::context::context (1 of 2 overloads)	1278
5.226.4.2	ssl::context::context (2 of 2 overloads)	1278
5.226.5	ssl::context::default_workarounds	1279
5.226.6	ssl::context::file_format	1279
5.226.7	ssl::context::load_verify_file	1279
5.226.7.1	ssl::context::load_verify_file (1 of 2 overloads)	1279
5.226.7.2	ssl::context::load_verify_file (2 of 2 overloads)	1280
5.226.8	ssl::context::method	1280
5.226.9	ssl::context::native_handle	1281
5.226.10	ssl::context::native_handle_type	1281

5.226.11	ssl::context::no_compression	1281
5.226.12	ssl::context::no_sslv2	1281
5.226.13	ssl::context::no_sslv3	1282
5.226.14	ssl::context::no_tlsv1	1282
5.226.15	ssl::context::no_tlsv1_1	1282
5.226.16	ssl::context::no_tlsv1_2	1282
5.226.17	ssl::context::operator=	1282
5.226.18	ssl::context::options	1282
5.226.19	ssl::context::password_purpose	1283
5.226.20	ssl::context::set_default_verify_paths	1283
5.226.20.1	ssl::context::set_default_verify_paths (1 of 2 overloads)	1283
5.226.20.2	ssl::context::set_default_verify_paths (2 of 2 overloads)	1283
5.226.21	ssl::context::set_options	1284
5.226.21.1	ssl::context::set_options (1 of 2 overloads)	1284
5.226.21.2	ssl::context::set_options (2 of 2 overloads)	1284
5.226.22	ssl::context::set_password_callback	1285
5.226.22.1	ssl::context::set_password_callback (1 of 2 overloads)	1285
5.226.22.2	ssl::context::set_password_callback (2 of 2 overloads)	1286
5.226.23	ssl::context::set_verify_callback	1286
5.226.23.1	ssl::context::set_verify_callback (1 of 2 overloads)	1286
5.226.23.2	ssl::context::set_verify_callback (2 of 2 overloads)	1287
5.226.24	ssl::context::set_verify_depth	1287
5.226.24.1	ssl::context::set_verify_depth (1 of 2 overloads)	1288
5.226.24.2	ssl::context::set_verify_depth (2 of 2 overloads)	1288
5.226.25	ssl::context::set_verify_mode	1288
5.226.25.1	ssl::context::set_verify_mode (1 of 2 overloads)	1289
5.226.25.2	ssl::context::set_verify_mode (2 of 2 overloads)	1289
5.226.26	ssl::context::single_dh_use	1289
5.226.27	ssl::context::use_certificate	1290
5.226.27.1	ssl::context::use_certificate (1 of 2 overloads)	1290
5.226.27.2	ssl::context::use_certificate (2 of 2 overloads)	1290
5.226.28	ssl::context::use_certificate_chain	1291
5.226.28.1	ssl::context::use_certificate_chain (1 of 2 overloads)	1291
5.226.28.2	ssl::context::use_certificate_chain (2 of 2 overloads)	1291
5.226.29	ssl::context::use_certificate_chain_file	1292
5.226.29.1	ssl::context::use_certificate_chain_file (1 of 2 overloads)	1292
5.226.29.2	ssl::context::use_certificate_chain_file (2 of 2 overloads)	1292
5.226.30	ssl::context::use_certificate_file	1293
5.226.30.1	ssl::context::use_certificate_file (1 of 2 overloads)	1293

5.226.30.2	ssl::context::use_certificate_file (2 of 2 overloads)	1293
5.226.31	ssl::context::use_private_key	1294
5.226.31.1	ssl::context::use_private_key (1 of 2 overloads)	1294
5.226.31.2	ssl::context::use_private_key (2 of 2 overloads)	1295
5.226.32	ssl::context::use_private_key_file	1295
5.226.32.1	ssl::context::use_private_key_file (1 of 2 overloads)	1295
5.226.32.2	ssl::context::use_private_key_file (2 of 2 overloads)	1296
5.226.33	ssl::context::use_rsa_private_key	1296
5.226.33.1	ssl::context::use_rsa_private_key (1 of 2 overloads)	1296
5.226.33.2	ssl::context::use_rsa_private_key (2 of 2 overloads)	1297
5.226.34	ssl::context::use_rsa_private_key_file	1297
5.226.34.1	ssl::context::use_rsa_private_key_file (1 of 2 overloads)	1297
5.226.34.2	ssl::context::use_rsa_private_key_file (2 of 2 overloads)	1298
5.226.35	ssl::context::use_tmp_dh	1298
5.226.35.1	ssl::context::use_tmp_dh (1 of 2 overloads)	1299
5.226.35.2	ssl::context::use_tmp_dh (2 of 2 overloads)	1299
5.226.36	ssl::context::use_tmp_dh_file	1299
5.226.36.1	ssl::context::use_tmp_dh_file (1 of 2 overloads)	1300
5.226.36.2	ssl::context::use_tmp_dh_file (2 of 2 overloads)	1300
5.226.37	ssl::context::~context	1300
5.227	ssl::context_base	1300
5.227.1	ssl::context_base::default_workarounds	1301
5.227.2	ssl::context_base::file_format	1302
5.227.3	ssl::context_base::method	1302
5.227.4	ssl::context_base::no_compression	1303
5.227.5	ssl::context_base::no_sslv2	1303
5.227.6	ssl::context_base::no_sslv3	1303
5.227.7	ssl::context_base::no_tlsv1	1303
5.227.8	ssl::context_base::no_tlsv1_1	1303
5.227.9	ssl::context_base::no_tlsv1_2	1303
5.227.10	ssl::context_base::options	1303
5.227.11	ssl::context_base::password_purpose	1304
5.227.12	ssl::context_base::single_dh_use	1304
5.227.13	ssl::context_base::~context_base	1304
5.228	ssl::error::get_stream_category	1304
5.229	ssl::error::make_error_code	1304
5.230	ssl::error::stream_category	1304
5.231	ssl::error::stream_errors	1305
5.232	ssl::rfc2818_verification	1305

5.232.1	ssl::rfc2818_verification::operator()	1306
5.232.2	ssl::rfc2818_verification::result_type	1306
5.232.3	ssl::rfc2818_verification::rfc2818_verification	1306
5.233	ssl::stream	1307
5.233.1	ssl::stream::async_handshake	1308
5.233.1.1	ssl::stream::async_handshake (1 of 2 overloads)	1309
5.233.1.2	ssl::stream::async_handshake (2 of 2 overloads)	1309
5.233.2	ssl::stream::async_read_some	1310
5.233.3	ssl::stream::async_shutdown	1310
5.233.4	ssl::stream::async_write_some	1311
5.233.5	ssl::stream::executor_type	1311
5.233.6	ssl::stream::get_executor	1311
5.233.7	ssl::stream::get_io_context	1312
5.233.8	ssl::stream::get_io_service	1312
5.233.9	ssl::stream::handshake	1312
5.233.9.1	ssl::stream::handshake (1 of 4 overloads)	1312
5.233.9.2	ssl::stream::handshake (2 of 4 overloads)	1313
5.233.9.3	ssl::stream::handshake (3 of 4 overloads)	1313
5.233.9.4	ssl::stream::handshake (4 of 4 overloads)	1313
5.233.10	ssl::stream::handshake_type	1314
5.233.11	ssl::stream::lowest_layer	1314
5.233.11.1	ssl::stream::lowest_layer (1 of 2 overloads)	1314
5.233.11.2	ssl::stream::lowest_layer (2 of 2 overloads)	1314
5.233.12	ssl::stream::lowest_layer_type	1315
5.233.13	ssl::stream::native_handle	1315
5.233.14	ssl::stream::native_handle_type	1315
5.233.15	ssl::stream::next_layer	1315
5.233.15.1	ssl::stream::next_layer (1 of 2 overloads)	1316
5.233.15.2	ssl::stream::next_layer (2 of 2 overloads)	1316
5.233.16	ssl::stream::next_layer_type	1316
5.233.17	ssl::stream::read_some	1316
5.233.17.1	ssl::stream::read_some (1 of 2 overloads)	1317
5.233.17.2	ssl::stream::read_some (2 of 2 overloads)	1317
5.233.18	ssl::stream::set_verify_callback	1318
5.233.18.1	ssl::stream::set_verify_callback (1 of 2 overloads)	1318
5.233.18.2	ssl::stream::set_verify_callback (2 of 2 overloads)	1319
5.233.19	ssl::stream::set_verify_depth	1319
5.233.19.1	ssl::stream::set_verify_depth (1 of 2 overloads)	1319
5.233.19.2	ssl::stream::set_verify_depth (2 of 2 overloads)	1320

5.233.20	ssl::stream::set_verify_mode	1320
5.233.20.1	ssl::stream::set_verify_mode (1 of 2 overloads)	1320
5.233.20.2	ssl::stream::set_verify_mode (2 of 2 overloads)	1321
5.233.21	ssl::stream::shutdown	1321
5.233.21.1	ssl::stream::shutdown (1 of 2 overloads)	1321
5.233.21.2	ssl::stream::shutdown (2 of 2 overloads)	1322
5.233.22	ssl::stream::stream	1322
5.233.23	ssl::stream::write_some	1322
5.233.23.1	ssl::stream::write_some (1 of 2 overloads)	1322
5.233.23.2	ssl::stream::write_some (2 of 2 overloads)	1323
5.233.24	ssl::stream::~stream	1323
5.234	ssl::stream::impl_struct	1324
5.234.1	ssl::stream::impl_struct::ssl	1324
5.235	ssl::stream_base	1324
5.235.1	ssl::stream_base::handshake_type	1325
5.235.2	ssl::stream_base::~stream_base	1325
5.236	ssl::verify_client_once	1325
5.237	ssl::verify_context	1325
5.237.1	ssl::verify_context::native_handle	1326
5.237.2	ssl::verify_context::native_handle_type	1326
5.237.3	ssl::verify_context::verify_context	1326
5.238	ssl::verify_fail_if_no_peer_cert	1326
5.239	ssl::verify_mode	1327
5.240	ssl::verify_none	1327
5.241	ssl::verify_peer	1327
5.242	steady_timer	1327
5.243	strand	1330
5.243.1	strand::context	1332
5.243.2	strand::defer	1332
5.243.3	strand::dispatch	1332
5.243.4	strand::get_inner_executor	1333
5.243.5	strand::inner_executor_type	1333
5.243.6	strand::on_work_finished	1333
5.243.7	strand::on_work_started	1333
5.243.8	strand::operator!=	1333
5.243.9	strand::operator=	1334
5.243.9.1	strand::operator= (1 of 4 overloads)	1334
5.243.9.2	strand::operator= (2 of 4 overloads)	1334
5.243.9.3	strand::operator= (3 of 4 overloads)	1334

5.243.9.4	strand::operator= (4 of 4 overloads)	1335
5.243.10	strand::operator==	1335
5.243.11	strand::post	1335
5.243.12	strand::running_in_this_thread	1335
5.243.13	strand::strand	1336
5.243.13.1	strand::strand (1 of 6 overloads)	1336
5.243.13.2	strand::strand (2 of 6 overloads)	1336
5.243.13.3	strand::strand (3 of 6 overloads)	1337
5.243.13.4	strand::strand (4 of 6 overloads)	1337
5.243.13.5	strand::strand (5 of 6 overloads)	1337
5.243.13.6	strand::strand (6 of 6 overloads)	1337
5.243.14	strand::~strand	1337
5.244	streambuf	1337
5.245	system_category	1339
5.246	system_context	1339
5.246.1	system_context::add_service	1341
5.246.2	system_context::destroy	1341
5.246.3	system_context::executor_type	1341
5.246.4	system_context::fork_event	1342
5.246.5	system_context::get_executor	1343
5.246.6	system_context::has_service	1343
5.246.7	system_context::join	1343
5.246.8	system_context::make_service	1343
5.246.9	system_context::notify_fork	1344
5.246.10	system_context::shutdown	1345
5.246.11	system_context::stop	1345
5.246.12	system_context::stopped	1345
5.246.13	system_context::use_service	1345
5.246.13.1	system_context::use_service (1 of 2 overloads)	1345
5.246.13.2	system_context::use_service (2 of 2 overloads)	1346
5.246.14	system_context::~system_context	1346
5.247	system_error	1347
5.247.1	system_error::code	1347
5.247.2	system_error::operator=	1347
5.247.3	system_error::system_error	1347
5.247.3.1	system_error::system_error (1 of 3 overloads)	1348
5.247.3.2	system_error::system_error (2 of 3 overloads)	1348
5.247.3.3	system_error::system_error (3 of 3 overloads)	1348
5.247.4	system_error::what	1348

5.247.5	system_error::~system_error	1348
5.248	system_executor	1348
5.248.1	system_executor::context	1349
5.248.2	system_executor::defer	1349
5.248.3	system_executor::dispatch	1350
5.248.4	system_executor::on_work_finished	1350
5.248.5	system_executor::on_work_started	1350
5.248.6	system_executor::operator!=	1350
5.248.7	system_executor::operator==	1351
5.248.8	system_executor::post	1351
5.249	system_timer	1351
5.250	thread	1354
5.250.1	thread::join	1355
5.250.2	thread::thread	1355
5.250.3	thread::~thread	1356
5.251	thread_pool	1356
5.251.1	thread_pool::add_service	1358
5.251.2	thread_pool::destroy	1358
5.251.3	thread_pool::fork_event	1358
5.251.4	thread_pool::get_executor	1359
5.251.5	thread_pool::has_service	1359
5.251.6	thread_pool::join	1359
5.251.7	thread_pool::make_service	1360
5.251.8	thread_pool::notify_fork	1360
5.251.9	thread_pool::shutdown	1361
5.251.10	thread_pool::stop	1361
5.251.11	thread_pool::thread_pool	1361
5.251.11.1	thread_pool::thread_pool (1 of 2 overloads)	1362
5.251.11.2	thread_pool::thread_pool (2 of 2 overloads)	1362
5.251.12	thread_pool::use_service	1362
5.251.12.1	thread_pool::use_service (1 of 2 overloads)	1362
5.251.12.2	thread_pool::use_service (2 of 2 overloads)	1363
5.251.13	thread_pool::~thread_pool	1363
5.252	thread_pool::executor_type	1363
5.252.1	thread_pool::executor_type::context	1364
5.252.2	thread_pool::executor_type::defer	1364
5.252.3	thread_pool::executor_type::dispatch	1365
5.252.4	thread_pool::executor_type::on_work_finished	1365
5.252.5	thread_pool::executor_type::on_work_started	1366

5.252.6	thread_pool::executor_type::operator!=	1366
5.252.7	thread_pool::executor_type::operator==	1366
5.252.8	thread_pool::executor_type::post	1366
5.252.9	thread_pool::executor_type::running_in_this_thread	1367
5.253	time_traits< boost::posix_time::ptime >	1367
5.253.1	time_traits< boost::posix_time::ptime >::add	1368
5.253.2	time_traits< boost::posix_time::ptime >::duration_type	1368
5.253.3	time_traits< boost::posix_time::ptime >::less_than	1368
5.253.4	time_traits< boost::posix_time::ptime >::now	1368
5.253.5	time_traits< boost::posix_time::ptime >::subtract	1368
5.253.6	time_traits< boost::posix_time::ptime >::time_type	1368
5.253.7	time_traits< boost::posix_time::ptime >::to_posix_duration	1369
5.254	transfer_all	1369
5.255	transfer_at_least	1369
5.256	transfer_exactly	1370
5.257	use_future	1371
5.258	use_future_t	1371
5.258.1	use_future_t::allocator_type	1372
5.258.2	use_future_t::get_allocator	1372
5.258.3	use_future_t::operator()	1372
5.258.4	use_future_t::operator[]	1373
5.258.5	use_future_t::rebind	1373
5.258.6	use_future_t::use_future_t	1373
5.258.6.1	use_future_t::use_future_t (1 of 2 overloads)	1373
5.258.6.2	use_future_t::use_future_t (2 of 2 overloads)	1373
5.259	use_service	1373
5.259.1	use_service (1 of 2 overloads)	1374
5.259.2	use_service (2 of 2 overloads)	1374
5.260	uses_executor	1375
5.261	wait_traits	1375
5.261.1	wait_traits::to_wait_duration	1375
5.261.1.1	wait_traits::to_wait_duration (1 of 2 overloads)	1375
5.261.1.2	wait_traits::to_wait_duration (2 of 2 overloads)	1376
5.262	windows::object_handle	1376
5.262.1	windows::object_handle::assign	1377
5.262.1.1	windows::object_handle::assign (1 of 2 overloads)	1377
5.262.1.2	windows::object_handle::assign (2 of 2 overloads)	1378
5.262.2	windows::object_handle::async_wait	1378
5.262.3	windows::object_handle::cancel	1378

5.262.3.1	windows::object_handle::cancel (1 of 2 overloads)	1378
5.262.3.2	windows::object_handle::cancel (2 of 2 overloads)	1379
5.262.4	windows::object_handle::close	1379
5.262.4.1	windows::object_handle::close (1 of 2 overloads)	1379
5.262.4.2	windows::object_handle::close (2 of 2 overloads)	1379
5.262.5	windows::object_handle::executor_type	1380
5.262.6	windows::object_handle::get_executor	1380
5.262.7	windows::object_handle::get_io_context	1380
5.262.8	windows::object_handle::get_io_service	1381
5.262.9	windows::object_handle::is_open	1381
5.262.10	windows::object_handle::lowest_layer	1381
5.262.10.1	windows::object_handle::lowest_layer (1 of 2 overloads)	1381
5.262.10.2	windows::object_handle::lowest_layer (2 of 2 overloads)	1382
5.262.11	windows::object_handle::lowest_layer_type	1382
5.262.12	windows::object_handle::native_handle	1383
5.262.13	windows::object_handle::native_handle_type	1383
5.262.14	windows::object_handle::object_handle	1384
5.262.14.1	windows::object_handle::object_handle (1 of 3 overloads)	1384
5.262.14.2	windows::object_handle::object_handle (2 of 3 overloads)	1384
5.262.14.3	windows::object_handle::object_handle (3 of 3 overloads)	1385
5.262.15	windows::object_handle::operator=	1385
5.262.16	windows::object_handle::wait	1385
5.262.16.1	windows::object_handle::wait (1 of 2 overloads)	1385
5.262.16.2	windows::object_handle::wait (2 of 2 overloads)	1386
5.263	windows::overlapped_handle	1386
5.263.1	windows::overlapped_handle::assign	1387
5.263.1.1	windows::overlapped_handle::assign (1 of 2 overloads)	1388
5.263.1.2	windows::overlapped_handle::assign (2 of 2 overloads)	1388
5.263.2	windows::overlapped_handle::cancel	1388
5.263.2.1	windows::overlapped_handle::cancel (1 of 2 overloads)	1388
5.263.2.2	windows::overlapped_handle::cancel (2 of 2 overloads)	1388
5.263.3	windows::overlapped_handle::close	1389
5.263.3.1	windows::overlapped_handle::close (1 of 2 overloads)	1389
5.263.3.2	windows::overlapped_handle::close (2 of 2 overloads)	1389
5.263.4	windows::overlapped_handle::executor_type	1389
5.263.5	windows::overlapped_handle::get_executor	1390
5.263.6	windows::overlapped_handle::get_io_context	1390
5.263.7	windows::overlapped_handle::get_io_service	1390
5.263.8	windows::overlapped_handle::is_open	1391

5.263.9	windows::overlapped_handle::lowest_layer	1391
5.263.9.1	windows::overlapped_handle::lowest_layer (1 of 2 overloads)	1391
5.263.9.2	windows::overlapped_handle::lowest_layer (2 of 2 overloads)	1391
5.263.10	windows::overlapped_handle::lowest_layer_type	1391
5.263.11	windows::overlapped_handle::native_handle	1393
5.263.12	windows::overlapped_handle::native_handle_type	1393
5.263.13	windows::overlapped_handle::operator=	1393
5.263.14	windows::overlapped_handle::overlapped_handle	1394
5.263.14.1	windows::overlapped_handle::overlapped_handle (1 of 3 overloads)	1394
5.263.14.2	windows::overlapped_handle::overlapped_handle (2 of 3 overloads)	1394
5.263.14.3	windows::overlapped_handle::overlapped_handle (3 of 3 overloads)	1395
5.263.15	windows::overlapped_handle::~overlapped_handle	1395
5.264	windows::overlapped_ptr	1395
5.264.1	windows::overlapped_ptr::complete	1396
5.264.2	windows::overlapped_ptr::get	1396
5.264.2.1	windows::overlapped_ptr::get (1 of 2 overloads)	1396
5.264.2.2	windows::overlapped_ptr::get (2 of 2 overloads)	1396
5.264.3	windows::overlapped_ptr::overlapped_ptr	1396
5.264.3.1	windows::overlapped_ptr::overlapped_ptr (1 of 2 overloads)	1397
5.264.3.2	windows::overlapped_ptr::overlapped_ptr (2 of 2 overloads)	1397
5.264.4	windows::overlapped_ptr::release	1397
5.264.5	windows::overlapped_ptr::reset	1397
5.264.5.1	windows::overlapped_ptr::reset (1 of 2 overloads)	1397
5.264.5.2	windows::overlapped_ptr::reset (2 of 2 overloads)	1398
5.264.6	windows::overlapped_ptr::~overlapped_ptr	1398
5.265	windows::random_access_handle	1398
5.265.1	windows::random_access_handle::assign	1399
5.265.1.1	windows::random_access_handle::assign (1 of 2 overloads)	1399
5.265.1.2	windows::random_access_handle::assign (2 of 2 overloads)	1400
5.265.2	windows::random_access_handle::async_read_some_at	1400
5.265.3	windows::random_access_handle::async_write_some_at	1401
5.265.4	windows::random_access_handle::cancel	1401
5.265.4.1	windows::random_access_handle::cancel (1 of 2 overloads)	1402
5.265.4.2	windows::random_access_handle::cancel (2 of 2 overloads)	1402
5.265.5	windows::random_access_handle::close	1402
5.265.5.1	windows::random_access_handle::close (1 of 2 overloads)	1402
5.265.5.2	windows::random_access_handle::close (2 of 2 overloads)	1403
5.265.6	windows::random_access_handle::executor_type	1403
5.265.7	windows::random_access_handle::get_executor	1404

5.265.8	windows::random_access_handle::get_io_context	1404
5.265.9	windows::random_access_handle::get_io_service	1404
5.265.10	windows::random_access_handle::is_open	1404
5.265.11	windows::random_access_handle::lowest_layer	1405
5.265.11.1	windows::random_access_handle::lowest_layer (1 of 2 overloads)	1405
5.265.11.2	windows::random_access_handle::lowest_layer (2 of 2 overloads)	1405
5.265.12	windows::random_access_handle::lowest_layer_type	1405
5.265.13	windows::random_access_handle::native_handle	1407
5.265.14	windows::random_access_handle::native_handle_type	1407
5.265.15	windows::random_access_handle::operator=	1407
5.265.16	windows::random_access_handle::random_access_handle	1407
5.265.16.1	windows::random_access_handle::random_access_handle (1 of 3 overloads)	1408
5.265.16.2	windows::random_access_handle::random_access_handle (2 of 3 overloads)	1408
5.265.16.3	windows::random_access_handle::random_access_handle (3 of 3 overloads)	1408
5.265.17	windows::random_access_handle::read_some_at	1409
5.265.17.1	windows::random_access_handle::read_some_at (1 of 2 overloads)	1409
5.265.17.2	windows::random_access_handle::read_some_at (2 of 2 overloads)	1410
5.265.18	windows::random_access_handle::write_some_at	1410
5.265.18.1	windows::random_access_handle::write_some_at (1 of 2 overloads)	1411
5.265.18.2	windows::random_access_handle::write_some_at (2 of 2 overloads)	1412
5.266	windows::stream_handle	1412
5.266.1	windows::stream_handle::assign	1413
5.266.1.1	windows::stream_handle::assign (1 of 2 overloads)	1414
5.266.1.2	windows::stream_handle::assign (2 of 2 overloads)	1414
5.266.2	windows::stream_handle::async_read_some	1414
5.266.3	windows::stream_handle::async_write_some	1415
5.266.4	windows::stream_handle::cancel	1416
5.266.4.1	windows::stream_handle::cancel (1 of 2 overloads)	1416
5.266.4.2	windows::stream_handle::cancel (2 of 2 overloads)	1416
5.266.5	windows::stream_handle::close	1416
5.266.5.1	windows::stream_handle::close (1 of 2 overloads)	1417
5.266.5.2	windows::stream_handle::close (2 of 2 overloads)	1417
5.266.6	windows::stream_handle::executor_type	1417
5.266.7	windows::stream_handle::get_executor	1418
5.266.8	windows::stream_handle::get_io_context	1418
5.266.9	windows::stream_handle::get_io_service	1418
5.266.10	windows::stream_handle::is_open	1419
5.266.11	windows::stream_handle::lowest_layer	1419
5.266.11.1	windows::stream_handle::lowest_layer (1 of 2 overloads)	1419

5.266.11.2	windows::stream_handle::lowest_layer (2 of 2 overloads)	1419
5.266.12	windows::stream_handle::lowest_layer_type	1419
5.266.13	windows::stream_handle::native_handle	1421
5.266.14	windows::stream_handle::native_handle_type	1421
5.266.15	windows::stream_handle::operator=	1421
5.266.16	windows::stream_handle::read_some	1422
5.266.16.1	windows::stream_handle::read_some (1 of 2 overloads)	1422
5.266.16.2	windows::stream_handle::read_some (2 of 2 overloads)	1423
5.266.17	windows::stream_handle::stream_handle	1423
5.266.17.1	windows::stream_handle::stream_handle (1 of 3 overloads)	1423
5.266.17.2	windows::stream_handle::stream_handle (2 of 3 overloads)	1424
5.266.17.3	windows::stream_handle::stream_handle (3 of 3 overloads)	1424
5.266.18	windows::stream_handle::write_some	1425
5.266.18.1	windows::stream_handle::write_some (1 of 2 overloads)	1425
5.266.18.2	windows::stream_handle::write_some (2 of 2 overloads)	1426
5.267	write	1426
5.267.1	write (1 of 12 overloads)	1428
5.267.2	write (2 of 12 overloads)	1429
5.267.3	write (3 of 12 overloads)	1430
5.267.4	write (4 of 12 overloads)	1431
5.267.5	write (5 of 12 overloads)	1432
5.267.6	write (6 of 12 overloads)	1433
5.267.7	write (7 of 12 overloads)	1434
5.267.8	write (8 of 12 overloads)	1435
5.267.9	write (9 of 12 overloads)	1436
5.267.10	write (10 of 12 overloads)	1437
5.267.11	write (11 of 12 overloads)	1437
5.267.12	write (12 of 12 overloads)	1438
5.268	write_at	1439
5.268.1	write_at (1 of 8 overloads)	1441
5.268.2	write_at (2 of 8 overloads)	1442
5.268.3	write_at (3 of 8 overloads)	1443
5.268.4	write_at (4 of 8 overloads)	1444
5.268.5	write_at (5 of 8 overloads)	1445
5.268.6	write_at (6 of 8 overloads)	1446
5.268.7	write_at (7 of 8 overloads)	1446
5.268.8	write_at (8 of 8 overloads)	1447
5.269	yield_context	1448
6	Networking TS compatibility	1450
7	Revision History	1455

List of Tables

1	AcceptableProtocol requirements	176
2	Buffer-oriented asynchronous random-access read device requirements	178
3	Buffer-oriented asynchronous random-access write device requirements	179
4	AsyncReadStream requirements	180
5	AsyncWriteStream requirements	180
6	CompletionCondition requirements	182
7	ConnectCondition requirements	184
8	ConstBufferSequence requirements	185
9	DynamicBuffer requirements	186
10	Endpoint requirements	188
11	Endpoint requirements for extensible implementations	188
12	EndpointSequence requirements	189
13	ExecutionContext requirements	190
14	Executor requirements	190
15	GettableSerialPortOption requirements	194
16	GettableSocketOption requirements for extensible implementations	194
17	Handler requirements	195
18	InternetProtocol requirements	197
19	IoControlCommand requirements for extensible implementations	197
20	IoObjectService requirements	198
21	MutableBufferSequence requirements	200
22	Protocol requirements	201
23	Protocol requirements for extensible implementations	202
24	SettableSerialPortOption requirements	206
25	SettableSocketOption requirements for extensible implementations	206
26	Buffer-oriented synchronous random-access read device requirements	209
27	Buffer-oriented synchronous random-access write device requirements	210
28	SyncReadStream requirements	211
29	SyncWriteStream requirements	212
30	TimeTraits requirements	212
31	WaitTraits requirements	215

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
 - Concurrency Hints
 - 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.

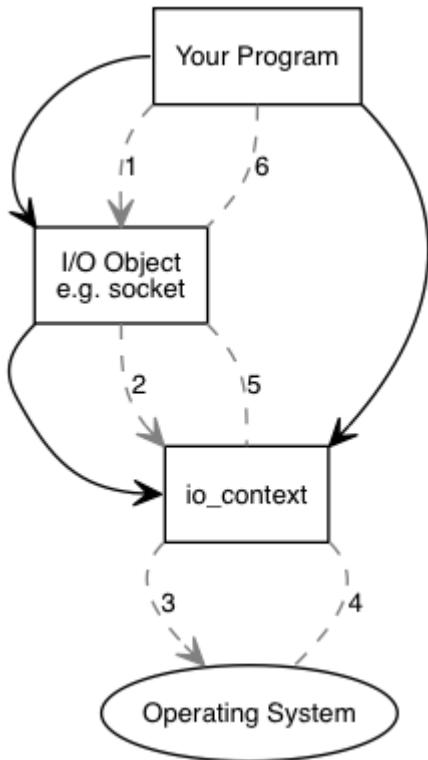
1.2 Core Concepts and Functionality

- 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
- Concurrency Hints
- 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_context** object. The **io_context** represents **your program**'s link to the **operating system**'s I/O services.

```
asio::io_context io_context;
```

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

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

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_context**.

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

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

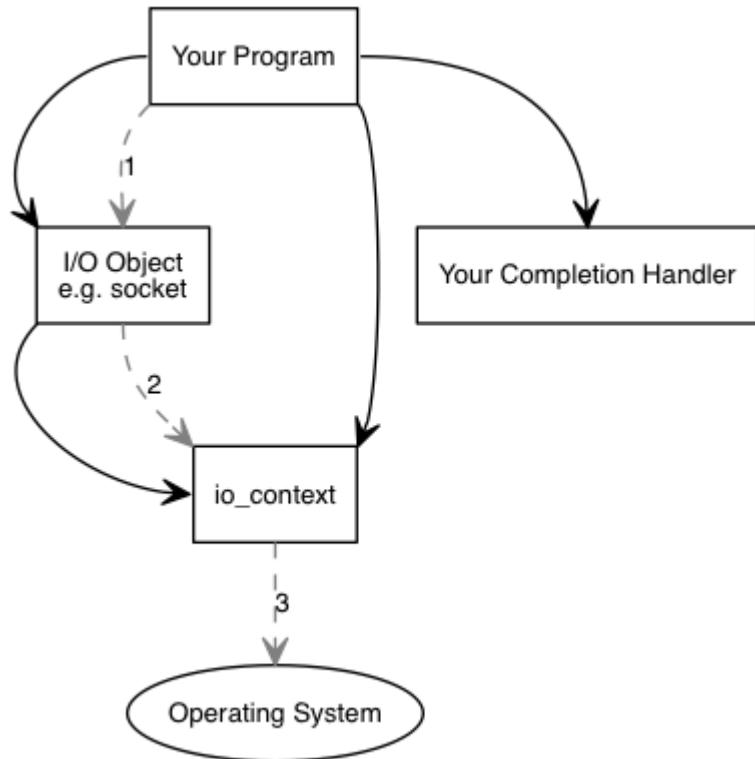
5. The **io_context** translates any error resulting from the operation into an object of type `asio::error_code`. An `error_code` may be compared with specific values, or tested as a boolean (where a `false` result means that no error occurred). The result is then forwarded back up to the **I/O object**.

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:

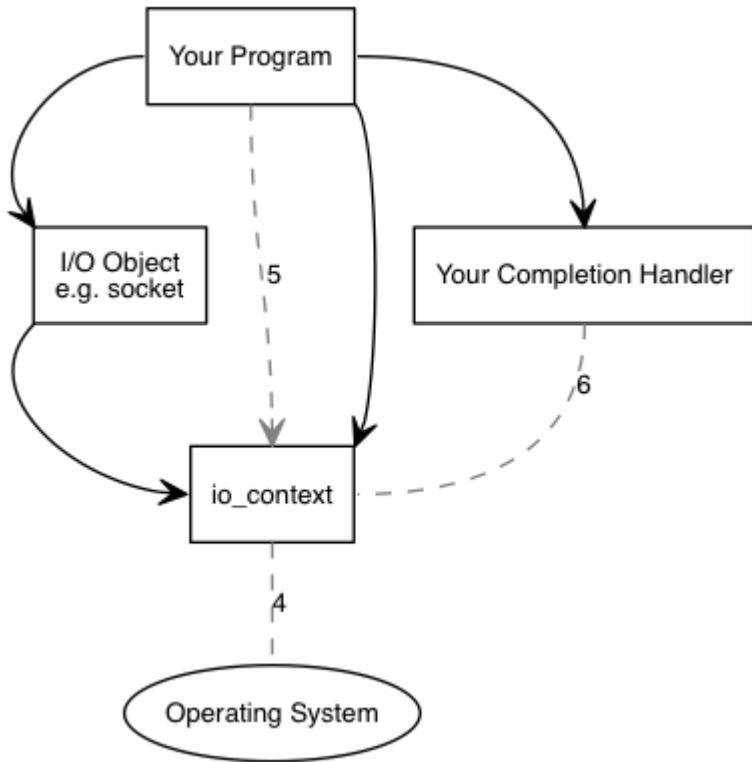
```
void your_completion_handler(const asio::error_code& ec);
```

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_context**.

3. The **io_context** 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_context**.
5. **Your program** must make a call to `io_context::run()` (or to one of the similar **io_context** member functions) in order for the result to be retrieved. A call to `io_context::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_context::run()`, the **io_context** dequeues the result of the operation, translates it into an `error_code`, and then passes it to **your completion handler**.

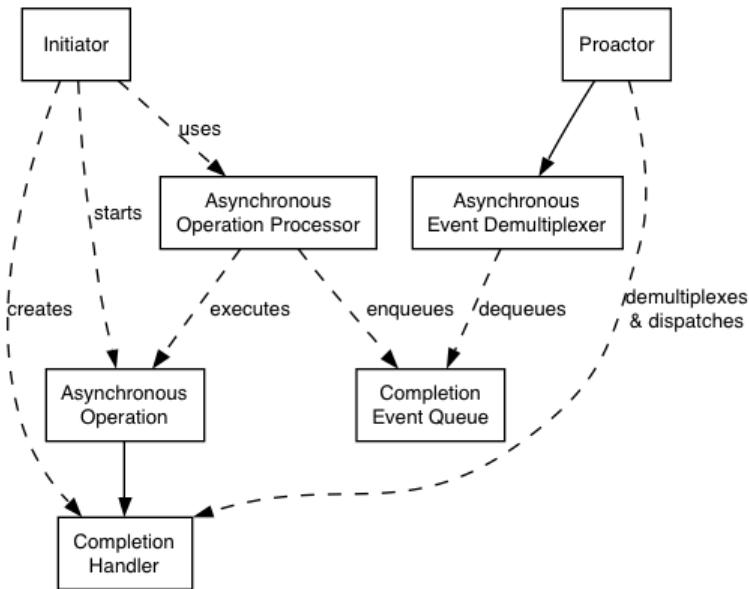
This is a simplified picture of how Asio operates. You will want to delve further into the documentation if your needs are more advanced, such as extending Asio to perform other types of asynchronous operations.

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_context` provide a stronger guarantee that it is safe to use a single object concurrently.

Thread Pools

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

Note that all threads that have joined an `io_context`'s pool are considered equivalent, and the `io_context` 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_context::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_context::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_context, post](#).

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_context::run()` from only one thread means all event handlers execute in an implicit strand, due to the `io_context`'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 `strand<>` or `io_context::strand`. All event handler function objects need to be bound to the strand using `asio::bind_executor()` or otherwise posted/dispatched through the 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).

To achieve this, all asynchronous operations obtain the handler's associated executor by using the `get_associated_executor` function. For example:

```
asio::associated_executor_t<Handler> a = asio::get_associated_executor(h);
```

The associated executor must satisfy the Executor requirements. It will be used by the asynchronous operation to submit both intermediate and final handlers for execution.

The executor may be customised for a particular handler type by specifying a nested type `executor_type` and member function `get_executor()`:

```
class my_handler
{
public:
    // Custom implementation of Executor type requirements.
    typedef my_executor executor_type;

    // Return a custom executor implementation.
    executor_type get_executor() const noexcept
    {
        return my_executor();
    }

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

In more complex cases, the `associated_executor` template may be partially specialised directly:

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

namespace asio {
```

```

template <class Executor>
struct associated_executor<my_handler, Executor>
{
    // Custom implementation of Executor type requirements.
    typedef my_executor type;

    // Return a custom executor implementation.
    static type get(const my_handler&,
                    const Executor& = Executor()) noexcept
    {
        return my_executor();
    }
};

} // namespaceasio

```

The `asio::bind_executor()` function is a helper to bind a specific executor object, such as a strand, to a completion handler. This binding automatically associates an executor as shown above. For example, to bind a strand to a completion handler we would simply write:

```

my_socket.async_read_some(my_buffer,
    asio::bind_executor(my_strand,
        [](error_code ec, size_t length)
    {
        // ...
    }));

```

See Also

[associated_executor](#), [get_associated_executor](#), [bind_executor](#), [strand](#), [io_context::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`.
- The underlying memory is explicitly accessed using the `data()` member 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 synchronous and asynchronous operations that may be used to wait for a socket to become ready to read, ready to write, or to have a pending error condition.

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

```
ip::tcp::socket socket(my_io_context);
...
socket.non_blocking(true);
...
socket.async_wait(ip::tcp::socket::wait_read, 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

[basic_socket::wait\(\)](#), [basic_socket::async_wait\(\)](#), [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 (c_ == *i++)
                return std::make_pair(i, true);
        return std::make_pair(i, false);
    }
}

```

```

private:
    char c_;
};

namespaceasio{
template<>structis_match_condition<match_char>
:publicboost::true_type{};
} //namespaceasio
...
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 obtain an allocator using the `get_associated_allocator` function. For example:

```
asio::associated_allocator_t<Handler> a = asio::get_associated_allocator(h);
```

The associated allocator must satisfy the standard Allocator requirements.

By default, handlers use the standard allocator (which is implemented in terms of `::operator new()` and `::operator delete()`). The allocator may be customised for a particular handler type by specifying a nested type `allocator_type` and member function `get_allocator()`:

```

classmy_handler
{
public:
    //CustomimplementationofAllocatortyperequirements.
    typedefmy_allocatorallocator_type;

    //Returnacustomallocatorimplementation.
    allocator_typeget_allocator()constnoexcept
    {
        returnmy_allocator();
    }

    voidoperator()() { ... }
};

```

In more complex cases, the `associated_allocator` template may be partially specialised directly:

```

namespace asio {

    template <typename Allocator>
    struct associated_allocator<my_handler, Allocator>
    {
        // Custom implementation of Allocator type requirements.
        typedef my_allocator type;

        // Return a custom allocator implementation.
        static type get(const my_handler&,
                        const Allocator& a = Allocator()) noexcept
        {
            return my_allocator();
        }
    };

} // namespace asio

```

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

[associated_allocator](#), [get_associated_allocator](#), [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|1512254357.979980|0*1|signal_set@0x7ffeaaaa20d8.async_wait
@asio|1512254357.980127|0*2|socket@0x7ffeaaaa20f8.async_accept
@asio|1512254357.980150|.2|non_blocking_accept,ec=asio.system:11
@asio|1512254357.980162|0|resolver@0x7ffeaaa1fd8.cancel
@asio|1512254368.457147|.2|non_blocking_accept,ec=system:0
@asio|1512254368.457193|>2|ec=system:0
@asio|1512254368.457219|2*3|socket@0x55cf39f0a238.async_receive
@asio|1512254368.457244|.3|non_blocking_recv,ec=system:0,bytes_transferred=141
@asio|1512254368.457275|2*4|socket@0x7ffeaaaa20f8.async_accept
@asio|1512254368.457293|.4|non_blocking_accept,ec=asio.system:11
@asio|1512254368.457301|<2|
@asio|1512254368.457310|>3|ec=system:0,bytes_transferred=141
@asio|1512254368.457441|3*5|socket@0x55cf39f0a238.async_send
@asio|1512254368.457502|.5|non_blocking_send,ec=system:0,bytes_transferred=156
@asio|1512254368.457511|<3|
@asio|1512254368.457519|>5|ec=system:0,bytes_transferred=156
@asio|1512254368.457544|5|socket@0x55cf39f0a238.close
@asio|1512254368.457559|<5|

```

```

@asio|1512254371.385106|>1|ec=system:0,signal_number=2
@asio|1512254371.385130|1|socket@0x7ffeaaaa20f8.close
@asio|1512254371.385163|<1|
@asio|1512254371.385175|>4|ec=asio.system:125
@asio|1512254371.385182|<4|
@asio|1512254371.385202|0|signal_set@0x7ffeaaaa20d8.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 `<timestamp>` 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_context` 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.
- .n** The implementation performed a system call as part of the asynchronous operation for which handler number `n` is the completion handler. The `<description>` shows what function was called and its results. These tracking events are only emitted when using a reactor-based implementation.

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`).

Custom Tracking

Handling tracking may be customised by defining the `ASIO_CUSTOM_HANDLER_TRACKING` macro to the name of a header file (enclosed in " " or <>). This header file must implement the following preprocessor macros:

Macro	Description
<code>ASIO_INHERIT_TRACKED_HANDLER</code>	Specifies a base class for classes that implement asynchronous operations. When used, the macro immediately follows the class name, so it must have the form : public my_class.

Macro	Description
ASIO_ALSO_INHERIT_TRACKED_HANDLER	Specifies a base class for classes that implement asynchronous operations. When used, the macro follows other base classes, so it must have the form , <code>public my_class</code> .
ASIO_HANDLER_TRACKING_INIT(args)	An expression that is used to initialise the tracking mechanism.
ASIO_HANDLER_CREATION(args)	An expression that is called on creation of an asynchronous operation. <code>args</code> is a parenthesised function argument list containing the owning execution context, the tracked handler, the name of the object type, a pointer to the object, the object's native handle, and the operation name.
ASIO_HANDLER_COMPLETION(args)	An expression that is called on completion of an asynchronous operation. <code>args</code> is a parenthesised function argument list containing the tracked handler.
ASIO_HANDLER_INVOCATION_BEGIN(args)	An expression that is called immediately before a completion handler is invoked. <code>args</code> is a parenthesised function argument list containing the arguments to the completion handler.
ASIO_HANDLER_INVOCATION_END	An expression that is called immediately after a completion handler is invoked.
ASIO_HANDLER_OPERATION(args)	An expression that is called when some synchronous object operation is called (such as <code>close()</code> or <code>cancel()</code>). <code>args</code> is a parenthesised function argument list containing the owning execution context, the name of the object type, a pointer to the object, the object's native handle, and the operation name.
ASIO_HANDLER_REACTOR_REGISTRATION(args)	An expression that is called when an object is registered with the reactor. <code>args</code> is a parenthesised function argument list containing the owning execution context, the object's native handle, and a unique registration key.
ASIO_HANDLER_REACTOR_DEREGISTRATION(args)	An expression that is called when an object is deregistered from the reactor. <code>args</code> is a parenthesised function argument list containing the owning execution context, the object's native handle, and a unique registration key.
ASIO_HANDLER_REACTOR_READ_EVENT	A bitmask constant used to identify reactor read readiness events.
ASIO_HANDLER_REACTOR_WRITE_EVENT	A bitmask constant used to identify reactor write readiness events.
ASIO_HANDLER_REACTOR_ERROR_EVENT	A bitmask constant used to identify reactor error readiness events.

Macro	Description
ASIO_HANDLER_REACTOR_EVENTS(args)	An expression that is called when an object registered with the reactor becomes ready. <code>args</code> is a parenthesised function argument list containing the owning execution context, the unique registration key, and a bitmask of the ready events.
ASIO_HANDLER_REACTOR_OPERATION(args)	An expression that is called when the implementation performs a system call as part of a reactor-based asynchronous operation. <code>args</code> is a parenthesised function argument list containing the tracked handler, the operation name, the error code produced by the operation, and (optionally) the number of bytes transferred.

See Also

[Custom handler tracking example](#).

1.2.11 Concurrency Hints

The `io_context` constructor allows programs to specify a concurrency hint. This is a suggestion to the `io_context` implementation as to the number of active threads that should be used for running completion handlers.

When the Windows I/O completion port backend is in use, this value is passed to `CreateIoCompletionPort`.

When a reactor-based backend is used, the implementation recognises the following special concurrency hint values:

Value	Description
1	The implementation assumes that the <code>io_context</code> will be run from a single thread, and applies several optimisations based on this assumption. For example, when a handler is posted from within another handler, the new handler is added to a fast thread-local queue (with the consequence that the new handler is held back until the currently executing handler finishes).
ASIO_CONCURRENCY_HINT_UNSAFE	This special concurrency hint disables locking in both the scheduler and reactor I/O. This hint has the following restrictions: — Care must be taken to ensure that all operations on the <code>io_context</code> and any of its associated I/O objects (such as sockets and timers) occur in only one thread at a time. — Asynchronous resolve operations fail with <code>operation_not_supported</code> . — If a <code>signal_set</code> is used with the <code>io_context</code> , <code>signal_set</code> objects cannot be used with any other <code>io_context</code> in the program.
ASIO_CONCURRENCY_HINT_UNSAFE_IO	This special concurrency hint disables locking in the reactor I/O. This hint has the following restrictions: — Care must be taken to ensure that run functions on the <code>io_context</code> , and all operations on the context's associated I/O objects (such as sockets and timers), occur in only one thread at a time.

Value	Description
ASIO_CONCURRENCY_HINT_SAFE	The default. The <code>io_context</code> provides full thread safety, and distinct I/O objects may be used from any thread.

The concurrency hint used by default-constructed `io_context` objects can be overridden at compile time by defining the `ASIO_CONCURRENCY_HINT` macro. For example, specifying

```
-DASIO_CONCURRENCY_HINT_DEFAULT=1
```

on the compiler command line means that a concurrency hint of 1 is used for all default-constructed `io_context` objects in the program. Similarly, the concurrency hint used by `io_context` objects constructed with 1 can be overridden by defining `ASIO_CONCURRENCY_HINT_1`. For example, passing

```
-DASIO_CONCURRENCY_HINT_1=ASIO_CONCURRENCY_HINT_UNSAFE
```

to the compiler will disable thread safety for all of these objects.

1.2.12 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](#).

1.2.13 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_context`, 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_context);
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_context);
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_context);
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_context, my_endpoint);
...
ip::tcp::socket socket(my_io_context);
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_context);
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_context, 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_context);
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_context, 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_context);
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_context);
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_context);
...
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_context);
...
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_context);
...
asio::generic::stream_protocol::socket my_socket(my_io_context);
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_context ioc;

ip::tcp::endpoint endpoint(tcp::v4(), 80);
ip::tcp::acceptor acceptor(ios, endpoint);

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

```

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_context::run()</code> , <code>io_context::run_one()</code> , <code>io_context::poll()</code> , <code>io_context::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_context 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_context 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](#), [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_context, 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](#), [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_context, 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_context);
local::stream_protocol::socket socket2(my_io_context);
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_context);
local::datagram_protocol::socket socket2(my_io_context);
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_context, ep);  
local::stream_protocol::socket socket(my_io_context);  
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_context);  
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.

These classes also provide limited support for regular files. This support assumes that the underlying read and write operations provided by the operating system never fail with `EAGAIN` or `EWOULDBLOCK`. (This assumption normally holds for buffered file I/O.) Synchronous and asynchronous read and write operations on file descriptors will succeed but the I/O will always be performed immediately. Wait operations, and operations involving `asio::null_buffers`, are not portably supported.

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_context, ::dup(STDIN_FILENO));  
posix::stream_descriptor out(my_io_context, ::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](#), [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_context.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_context_.notify_fork(asio::io_context::fork_prepare);
if (fork() == 0)
{
    io_context_.notify_fork(asio::io_context::fork_child);
    ...
}
else
{
    io_context_.notify_fork(asio::io_context::fork_parent);
    ...
}
```

User-defined services can also be made fork-aware by overriding the `io_context::service::notify_fork()` 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_context::notify_fork\(\)](#), [io_context::fork_event](#), [io_context::service::notify_fork\(\)](#), [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_context, 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](#).

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_context, 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](#).

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_context, 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](#).

Notes

Windows object `HANDLEs` are only available at compile time when targeting Windows. Programs may test for the macro `ASIO_HAS_WINDOWS` 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_context, 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_context);
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_context io_context;
ssl_socket sock(io_context, ctx);
tcp::resolver resolver(io_context);
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_context& 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_;
    ...
void handle_accept(error_code ec, tcp::socket socket)
{
    if (!ec)
        std::make_shared<connection>(std::move(socket))->go();
    acceptor_.async_accept(...);
}
...
};
```

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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` 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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` 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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` 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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` 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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` 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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` 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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` 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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` 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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` 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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` 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_context::run()`, `io_context::run_one()`, `io_context::poll()` or `io_context::poll_one()`.
- An additional thread per `io_context` is used to trigger timers. This thread is created on construction of the first `basic_deadline_timer` or `basic_waitable_timer` objects.
- An additional thread per `io_context` is used for the `select` demultiplexing. This thread is created on the first call to `async_connect()`.
- An additional thread per `io_context` 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_context`, `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_context` for the handler to be executed.
- An additional thread per `io_context` 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 platform and compiler combinations are regularly tested:

- Linux using g++ 4.1 or later
- Linux using clang 3.2 or later
- FreeBSD using g++ 4.1 or later
- macOS using Xcode 8 or later
- Win32 using Visual C++ 9.0 or later
- Win32 using g++ 4.1 or later (MinGW)
- Win64 using Visual C++ 9.0 or later

The following platforms may also work:

- AIX
- Android
- HP-UX
- iOS
- NetBSD
- OpenBSD
- QNX Neutrino
- Solaris
- Tru64
- Win32 using Cygwin. (`__USE_W32_SOCKETS` must be defined.)

Dependencies

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

- Boost.Coroutine (optional) if you use `spawn()` to launch coroutines.
- 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 9.0 (or later) 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.

Macros

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

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.

Macro	Description
ASIO_ENABLE_BUFFER_DEBUGGING	Enables Asio's buffer debugging support, which can help identify when invalid buffers are used in read or write operations (e.g. if a <code>std::string</code> object being written is destroyed before the write operation completes). When using Microsoft Visual C++ 11.0 or later, this macro is defined automatically if the compiler's iterator debugging support is enabled, unless <code>ASIO_DISABLE_BUFFER_DEBUGGING</code> has been defined. When using <code>g++</code> , this macro is defined automatically if standard library debugging is enabled (<code>_GLIBCXX_DEBUG</code> is defined), unless <code>ASIO_DISABLE_BUFFER_DEBUGGING</code> has been defined.
ASIO_DISABLE_BUFFER_DEBUGGING	Explicitly disables Asio's buffer debugging support.
ASIO_DISABLE_DEV_POLL	Explicitly disables <code>/dev/poll</code> support on Solaris, forcing the use of a <code>select</code> -based implementation.
ASIO_DISABLE_EPOLL	Explicitly disables <code>epoll</code> support on Linux, forcing the use of a <code>select</code> -based implementation.
ASIO_DISABLE_EVENTFD	Explicitly disables <code>eventfd</code> support on Linux, forcing the use of a pipe to interrupt blocked <code>epoll/select</code> system calls.
ASIO_DISABLE_KQUEUE	Explicitly disables <code>kqueue</code> support on macOS and BSD variants, forcing the use of a <code>select</code> -based implementation.
ASIO_DISABLE_IOCP	Explicitly disables I/O completion ports support on Windows, forcing the use of a <code>select</code> -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	By default, Asio will automatically define <code>WIN32_LEAN_AND_MEAN</code> when compiling for Windows, to minimise the number of Windows SDK header files and features that are included. The presence of <code>ASIO_NO_WIN32_LEAN_AND_MEAN</code> prevents <code>WIN32_LEAN_AND_MEAN</code> from being defined.
ASIO_NO_NOMINMAX	By default, Asio will automatically define <code>NOMINMAX</code> when compiling for Windows, to suppress the definition of the <code>min()</code> and <code>max()</code> macros. The presence of <code>ASIO_NO_NOMINMAX</code> prevents <code>NOMINMAX</code> from being defined.

Macro	Description
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. <code>ws2_32.lib</code> and <code>mswsock.lib</code>, or <code>ws2.lib</code> when building for Windows CE). The <code>ASIO_NO_DEFAULT_LINKED_LIBS</code> macro prevents these libraries from being linked.</p>
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	<p>Disables uses of the <code>typeid</code> operator in asio. Defined automatically if <code>BOOST_NO_TYPEID</code> is defined.</p>
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.

Macro	Description
ASIO_ENABLE_OLD_SERVICES	The service template parameters, and the corresponding classes, are disabled by default. For example, instead of <code>basic_socket<Protocol, SocketService></code> we now have simply <code>basic_socket<Protocol></code> . The old interface can be enabled by defining the ASIO_ENABLE_OLD_SERVICES macro.

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_context` 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_context 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_context` 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) 2003-2017 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_context 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_context 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_context::run()` member function on the `io_context` object.

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

The `io_context::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_context` some work to do before calling `io_context::run()`. For example, if we had omitted the above call to `deadline_timer::async_wait()`, the `io_context` would not have had any work to do, and consequently `io_context::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) 2003-2017 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_context 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_context` to stop. Recall that in tutorial Timer.2 we learnt that the `io_context::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_context` 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_context 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) 2003-2017 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_context 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_context` 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_context& 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_context` as normal.

```

int main()
{
    asio::io_context 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) 2003-2017 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_context& 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_context 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 `io_context::strand` class to synchronise callback handlers in a multithreaded program.

The previous four tutorials avoided the issue of handler synchronisation by calling the `io_context::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_context::run()`. Consequently, calling `io_context::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_context::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 `io_context::strand`.

An `io_context::strand` is an executor that 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_context::run()`. Of course, the handlers may still execute concurrently with other handlers that were not dispatched through an `io_context::strand`, or were dispatched through a different `io_context::strand` object.

```
printer(asio::io_context& 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 "bound" to an `io_context::strand` object. The `asio::io_context::strand::bind` function returns a new handler that automatically dispatches its contained handler through the `io_context::strand` object. By binding the handlers to the same `io_context::strand`, we are ensuring that they cannot execute concurrently.

```
timer1_.async_wait(asio::bind_executor(strand_,
    boost::bind(&printer::print1, this)));

timer2_.async_wait(asio::bind_executor(strand_,
    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(asio::bind_executor(strand_,
            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(asio::bind_executor(strand_,
            boost::bind(&printer::print2, this)));
    }
}

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

The `main` function now causes `io_context::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_context::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_context io;
    printer p(io);
    asio::thread t(boost::bind(&asio::io_context::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) 2003-2017 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_context& io)
        : strand_(io),
          timer1_(io, boost::posix_time::seconds(1)),
          timer2_(io, boost::posix_time::seconds(1)),
          count_(0)
    {
        timer1_.async_wait(asio::bind_executor(strand_,
                                               boost::bind(&printer::print1, this)));
        timer2_.async_wait(asio::bind_executor(strand_,
                                               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(asio::bind_executor(strand_,
                                                   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(asio::bind_executor(strand_,
                                                   boost::bind(&printer::print2, this)));
        }
    }
};

```

```

    }
}

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

int main()
{
    asio::io_context io;
    printer p(io);
    asio::thread t(boost::bind(&asio::io_context::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_context` object.

```
asio::io_context io_context;
```

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_context);
```

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".

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.)

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.

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`.

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.

Finally, handle any exceptions that may have been thrown.

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) 2003-2017 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_context io_context;  
  
        tcp::resolver resolver(io_context);  
        tcp::resolver::results_type endpoints =  
            resolver.resolve(argv[1], "daytime");  
  
        tcp::socket socket(io_context);  
        asio::connect(socket, endpoints);  
  
        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_context io_context;

```

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_context, 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_context);
        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) 2003-2017 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_context io_context;

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

        for (;;)
        {

```

```

        tcp::socket socket(io_context);
        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_context` object provides I/O services, such as sockets, that the server object will use.

```

asio::io_context io_context;
tcp_server server(io_context);

```

Run the `io_context` object so that it will perform asynchronous operations on your behalf.

```

    io_context.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_context& io_context)
    : acceptor_(io_context, 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_executor().context());

    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_context& io_context)
    {
        return pointer(new tcp_connection(io_context));
    }

    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_, asio::buffer(message_),
    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_context& io_context)
    : socket_(io_context)
{}
```

```
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) 2003-2017 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_context& io_context)
    {
        return pointer(new tcp_connection(io_context));
    }

    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_context& io_context)
        : socket_(io_context)
    {}

    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_context& io_context)
        : acceptor_(io_context, tcp::endpoint(tcp::v4(), 13))
    {
        start_accept();
    }

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

        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_context io_context;
        tcp_server server(io_context);
        io_context.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_context io_context;
```

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

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::socket socket(io_context);
        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;
```

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.

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\(\)](#).

Finally, handle any exceptions that may have been thrown.

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) 2003-2017 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_context io_context;

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

        udp::socket socket(io_context);
        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_context io_context;

```

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

```
    udp::socket socket(io_context, 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\(\)](#).

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

Send the response to the `remote_endpoint`.

Finally, handle any exceptions.

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) 2003-2017 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_context io_context;  
  
        udp::socket socket(io_context, 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);  
  
            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_context` object.

```

        asio::io_context io_context;
        udp_server server(io_context);
        io_context.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_context& io_context)
        : socket_(io_context, 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_context` 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_context` object would return an error if the client sent anything larger. We can ignore such an error if it comes up.

```

if (!error)
{
}

```

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) 2003-2017 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_context& io_context)  
        : socket_(io_context, 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)  
        {  
            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,  
                    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_context io_context;
        udp_server server(io_context);
        io_context.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_context io_context;

```

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

```
    tcp_server server1(io_context);
```

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

```
    udp_server server2(io_context);
```

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

```

        io_context.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_context& io_context)
    {
        return pointer(new tcp_connection(io_context));
    }

    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_context& io_context)
        : socket_(io_context)
    {}

    void handle_write()
    {}

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

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

private:
    void start_accept()
    {
        tcp_connection::pointer new_connection =

```

```

    tcp_connection::create(acceptor_.get_executor().context());

    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_context& io_context)
        : socket_(io_context, 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)
        {
            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) 2003-2017 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_context& io_context)
    {
        return pointer(new tcp_connection(io_context));
    }

    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_context& io_context)
        : socket_(io_context)
    {
    }

    void handle_write()
    {
    }

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

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

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

        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_context& io_context)
        : socket_(io_context, 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)
    {
        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_context io_context;
    tcp_server server1(io_context);
    udp_server server2(io_context);
    io_context.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_context-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_context_pool.cpp`
- `./src/examples/cpp03/http/server2/io_context_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_context and a thread pool calling `io_context::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_context`, 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/iostream_client.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](#) (diff to C++03)

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)

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/cpp11/fork/daemon.cpp](#) (diff to C++03)

The second example demonstrates how it is possible to fork a process from within a completion handler.

- [..../src/examples/cpp11/fork/process_per_connection.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](#)

Handler Tracking

This example shows how to implement custom handler tracking.

- [./src/examples/cpp11/handler_tracking/custom_tracking.hpp](#)

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 \(diff to C++03\)](#)

Multicast

An example showing the use of multicast to transmit packets to a group of subscribers.

- [./src/examples/cpp11/multicast/receiver.cpp \(diff to C++03\)](#)
- [./src/examples/cpp11/multicast/sender.cpp \(diff to C++03\)](#)

Nonblocking

Example demonstrating reactor-style operations for integrating a third-party library that wants to perform the I/O operations itself.

- [./src/examples/cpp11/nonblocking/third_party_lib.cpp](#) (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)

UNIX Domain Sockets

Examples showing how to use UNIX domain (local) sockets.

- [./src/examples/cpp11/local/connect_pair.cpp](#) (diff to C++03)
- [./src/examples/cpp11/local/iostream_client.cpp](#) (diff to C++03)
- [./src/examples/cpp11/local/stream_server.cpp](#) (diff to C++03)
- [./src/examples/cpp11/local/stream_client.cpp](#) (diff to C++03)

5 Reference

Core			
Classes	Free Functions	Class Templates	Error Codes
Classes <pre>coroutine error_code execution_context execution_context::id execution_context::service executor executor_arg_t invalid_service_owner io_context io_context::executor_type io_context::service io_context::strand io_context::work (deprecated) service_already_exists system_error system_executor thread thread_pool thread_pool::executor_type yield_context</pre>	Free Functions <pre>add_service asio_handler_allocate asio_handler_deallocate asio_handler_invoke asio_handler_is_continuation bind_executor dispatch defer get_associated_allocator get_associated_executor has_service make_work_guard post spawn use_service</pre>	Class Templates <pre>async_completion basic_io_object basic_yield_context executor_binder executor_work_guard strand use_future_t</pre> Special Values <pre>executor_arg use_future</pre> Boost.Bind Placeholders <pre>placeholders::bytes_transferred placeholders::endpoint placeholders::error placeholders::iterator placeholders::results placeholders::signal_number</pre>	Error Codes <pre>error::basic_errors error::netdb_errors error::addrinfo_errors error::misc_errors</pre> Type Traits <pre>associated_allocator associated_executor async_result handler_type (deprecated) is_executor uses_executor</pre> Type Requirements <pre>Asynchronous operations CompletionHandler ExecutionContext Executor Handler Service</pre>

Buffers and Buffer-Oriented Operations			
Classes	Free Functions	Type Traits	Type Requirements
Classes <pre>const_buffer mutable_buffer const_buffers_1 (deprecated) mutable_buffers_1 (deprecated) null_buffers (deprecated) streambuf</pre> Class Templates <pre>basic_streambuf buffered_read_stream buffered_stream buffered_write_stream buffers_iterator dynamic_string_buffer dynamic_vector_buffer</pre>	Free Functions <pre>async_read async_read_at async_read_until async_write async_write_at buffer buffer_cast (deprecated) buffer_copy buffer_size buffers_begin buffers_end dynamic_buffer read read_at read_until transfer_all transfer_at_least transfer_exactly write write_at</pre>	Type Traits <pre>is_const_buffer_sequence is_match_condition is_mutable_buffer_sequence is_read_buffered is_write_buffered</pre>	Type Requirements <pre>Read and write operations AsyncRandomAccessReadDevice AsyncRandomAccessWriteDevice AsyncReadStream AsyncWriteStream CompletionCondition ConstBufferSequence DynamicBuffer MutableBufferSequence ReadHandler SyncRandomAccessReadDevice SyncRandomAccessWriteDevice SyncReadStream SyncWriteStream WriteHandler</pre>

Networking			
Classes	Free Functions	Socket Options	I/O Control Commands
Classes 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_v4_iterator ip::address_v4_range ip::address_v6 ip::address_v6_iterator ip::address_v6_range ip::icmp ip::icmp::endpoint ip::icmp::resolver ip::icmp::socket ip::network_v4 ip::network_v6 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	Free Functions async_connect connect ip::host_name ip::make_address ip::make_address_v4 ip::make_address_v6 ip::make_network_v4 ip::make_network_v6 Class Templates basic_datagram_socket basic_raw_socket basic_seq_packet_socket basic_socket basic_socket_acceptor basic_socket_iostream basic_socket_streampub basic_stream_socket generic::basic_endpoint ip::basic_endpoint ip::basic_resolver ip::basic_resolver_entry ip::basic_resolver_iterator ip::basic_resolver_query	Socket Options 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	I/O Control Commands socket_base::bytes_readable Type Requirements Synchronous socket operations Asynchronous socket operations AcceptableProtocol AcceptHandler ConnectCondition ConnectHandler Endpoint EndpointSequence GettableSocketOption InternetProtocol IoControlCommand IteratorConnectHandler MoveAcceptHandler Protocol RangeConnectHandler ResolveHandler SettableSocketOption

Timers	SSL	Serial Ports	Signal Handling
Classes deadline_timer high_resolution_timer steady_timer system_timer Class Templates basic_deadline_timer basic_writable_timer time_traits wait_traits Type Requirements TimeTraits WaitHandler WaitTraits	Classes ssl::context ssl::context_base ssl::rfc2818_verification ssl::stream_base ssl::verify_context Class Templates ssl::stream Type Requirements BufferedHandshakeHandler HandshakeHandler ShutdownHandler	Classes serial_port serial_port_base Serial Port Options serial_port_base::baud_rate serial_port_base::flow_control serial_port_base::parity serial_port_base::stop_bits serial_port_base::character_size Type Requirements GettableSerialPortOption SettableSerialPortOption	Classes signal_set Type Requirements SignalHandler

POSIX-specific		Windows-specific
Classes	Free Functions	Classes
Classes <code>local::stream_protocol</code> <code>local::stream_protocol::acceptor</code> <code>local::stream_protocol::endpoint</code> <code>local::stream_protocol::iostream</code> <code>local::stream_protocol::socket</code> <code>local::datagram_protocol</code> <code>local::datagram_protocol::endpoint</code> <code>local::datagram_protocol::socket</code> <code>posix::descriptor</code> <code>posix::descriptor_base</code> <code>posix::stream_descriptor</code>	Free Functions <code>local::connect_pair</code> Class Templates <code>local::basic_endpoint</code>	Classes <code>windows::object_handle</code> <code>windows::overlapped_handle</code> <code>windows::overlapped_ptr</code> <code>windows::random_access_handle</code> <code>windows::stream_handle</code>

5.1 Requirements on asynchronous operations

This section uses the names `Alloc1`, `Alloc2`, `alloc1`, `alloc2`, `Args`, `CompletionHandler`, `completion_handler`, `Executor1`, `Executor2`, `ex1`, `ex2`, `f`, `i`, `N`, `Signature`, `token`, `T`[_i], `t`[_i], `work1`, and `work2` as placeholders for specifying the requirements below.

5.1.1 General asynchronous operation concepts

An *initiating function* is a function which may be called to start an asynchronous operation. A *completion handler* is a function object that will be invoked, at most once, with the result of the asynchronous operation.

The lifecycle of an asynchronous operation is comprised of the following events and phases:

- Event 1: The asynchronous operation is started by a call to the initiating function.
- Phase 1: The asynchronous operation is now *outstanding*.
- Event 2: The externally observable side effects of the asynchronous operation, if any, are fully established. The completion handler is submitted to an executor.
- Phase 2: The asynchronous operation is now *completed*.
- Event 3: The completion handler is called with the result of the asynchronous operation.

In this library, all functions with the prefix `async_` are initiating functions.

5.1.2 Completion tokens and handlers

Initiating functions:

- are function templates with template parameter `CompletionToken`;
- accept, as the final parameter, a *completion token* object `token` of type `CompletionToken`;
- specify a *completion signature*, which is a call signature (C++Std [func.def]) `Signature` that determines the arguments to the completion handler.

An initiating function determines the type `CompletionHandler` of its completion handler function object by performing `typename async_result<decay_t<CompletionToken>, Signature>::completion_handler_type`. The completion handler object `completion_handler` is initialized with `forward<CompletionToken>(token)`. [Note: No other requirements are placed on the type `CompletionToken`. —end note]

The type `CompletionHandler` must satisfy the requirements of `Destructible` (C++Std [destructible]) and `MoveConstructible` (C++Std [moveconstructible]), and be callable with the specified call signature.

In this library, all initiating functions specify a *Completion signature* element that defines the call signature `Signature`. The *Completion signature* elements in this Technical Specification have named parameters, and the results of an asynchronous operation are specified in terms of these names.

5.1.3 Automatic deduction of initiating function return type

The return type of an initiating function is `typename async_result<decay_t<CompletionToken>, Signature>::return_type`.

For the sake of exposition, this library sometimes annotates functions with a return type *DEDUCED*. For every function declaration that returns *DEDUCED*, the meaning is equivalent to specifying the return type as `typename async_result<decay_t<CompletionToken>, Signature>::return_type`.

5.1.4 Production of initiating function return value

An initiating function produces its return type as follows:

- constructing an object `result` of type `async_result<decay_t<CompletionToken>, Signature>`, initialized as `result(completion_handler);` and
- using `result.get()` as the operand of the return statement.

[*Example*: Given an asynchronous operation with *Completion signature* `void(R1 r1, R2 r2)`, an initiating function meeting these requirements may be implemented as follows:

```
template<class CompletionToken>
auto async_xyz(T1 t1, T2 t2, CompletionToken&& token)
{
    typename async_result<decay_t<CompletionToken>, void(R1, R2)>::completion_handler_type
        completion_handler(forward<CompletionToken>(token));

    async_result<decay_t<CompletionToken>, void(R1, R2)> result(completion_handler);

    // initiate the operation and cause completion_handler to be invoked with
    // the result

    return result.get();
}
```

For convenience, initiating functions may be implemented using the `async_completion` template:

```
template<class CompletionToken>
auto async_xyz(T1 t1, T2 t2, CompletionToken&& token)
{
    async_completion<CompletionToken, void(R1, R2)> init(token);

    // initiate the operation and cause init.completion_handler to be invoked
    // with the result

    return init.result.get();
}
```

—*end example*]

5.1.5 Lifetime of initiating function arguments

Unless otherwise specified, the lifetime of arguments to initiating functions shall be treated as follows:

- If the parameter has a pointer type or has a type of lvalue reference to non-const, the implementation may assume the validity of the pointee or referent, respectively, until the completion handler is invoked. [*Note*: In other words, the program must guarantee the validity of the argument until the completion handler is invoked. —*end note*]
- Otherwise, the implementation must not assume the validity of the argument after the initiating function completes. [*Note*: In other words, the program is not required to guarantee the validity of the argument after the initiating function completes. —*end note*] The implementation may make copies of the argument, and all copies shall be destroyed no later than immediately after invocation of the completion handler.

5.1.6 Non-blocking requirements on initiating functions

An initiating function shall not block (C++Std [defns.block]) the calling thread pending completion of the outstanding operation.

[*std_note* Initiating functions may still block the calling thread for other reasons. For example, an initiating function may lock a mutex in order to synchronize access to shared data.]

5.1.7 Associated executor

Certain objects that participate in asynchronous operations have an *associated executor*. These are obtained as specified below.

5.1.8 I/O executor

An asynchronous operation has an associated executor satisfying the [Executor](#) requirements. If not otherwise specified by the asynchronous operation, this associated executor is an object of type `system_executor`.

All asynchronous operations in this library have an associated executor object that is determined as follows:

- If the initiating function is a member function, the associated executor is that returned by the `get_executor` member function on the same object.
- If the initiating function is not a member function, the associated executor is that returned by the `get_executor` member function of the first argument to the initiating function.

Let `Executor1` be the type of the associated executor. Let `ex1` be a value of type `Executor1`, representing the associated executor object obtained as described above.

5.1.9 Completion handler executor

A completion handler object of type `CompletionHandler` has an associated executor of type `Executor2` satisfying the [Executor requirements](#). The type `Executor2` is `associated_executor_t<CompletionHandler, Executor1>`. Let `ex2` be a value of type `Executor2` obtained by performing `get_associated_executor(completion_handler, ex1)`.

5.1.10 Outstanding work

Until the asynchronous operation has completed, the asynchronous operation shall maintain:

- an object `work1` of type `executor_work_guard<Executor1>`, initialized as `work1(ex1)`, and where `work1.owns_work() == true`; and
- an object `work2` of type `executor_work_guard<Executor2>`, initialized as `work2(ex2)`, and where `work2.owns_work() == true`.

5.1.11 Allocation of intermediate storage

Asynchronous operations may allocate memory. [*Note*: Such as a data structure to store copies of the `completion_handler` object and the initiating function's arguments. —*end note*]

Let `Alloc1` be a type, satisfying the [ProtoAllocator](#) requirements, that represents the asynchronous operation's default allocation strategy. [*Note*: Typically `std::allocator<void>`. —*end note*] Let `alloc1` be a value of type `Alloc1`.

A completion handler object of type `CompletionHandler` has an associated allocator object `alloc2` of type `Alloc2` satisfying the [ProtoAllocator](#) requirements. The type `Alloc2` is `associated_allocator_t<CompletionHandler, Alloc1>`. Let `alloc2` be a value of type `Alloc2` obtained by performing `get_associated_allocator(completion_handler, alloc1)`.

The asynchronous operations defined in this library:

- If required, allocate memory using only the completion handler's associated allocator.

- Prior to completion handler execution, deallocate any memory allocated using the completion handler's associated allocator.
- [*std_note* The implementation may perform operating system or underlying API calls that perform memory allocations not using the associated allocator. Invocations of the allocator functions may not introduce data races (See C++Std [res.on.data.races].)]

5.1.12 Execution of completion handler on completion of asynchronous operation

Let *Args*... be the argument types of the completion signature *Signature* and let *N* be `sizeof...`(*Args*). Let *i* be in the range [0,*N*). Let *T*[_{sub} *i*] be the *i*th type in *Args*... and let *t*[_{sub} *i*] be the *i*th completion handler argument associated with *T*[_{sub} *i*].

Let *f* be a function object, callable as *f*(), that invokes `completion_handler` as if by `completion_handler(forward<T[sub 0]>(t[sub 0]), ..., forward<T[sub N-1]>(t[sub N-1]))`.

If an asynchronous operation completes immediately (that is, within the thread of execution calling the initiating function, and before the initiating function returns), the completion handler shall be submitted for execution as if by performing `ex2.post(std::move(f), alloc2)`. Otherwise, the completion handler shall be submitted for execution as if by performing `ex2.dispatch(std::move(f), alloc2)`.

5.1.13 Completion handlers and exceptions

Completion handlers are permitted to throw exceptions. The effect of any exception propagated from the execution of a completion handler is determined by the executor which is executing the completion handler.

5.2 Requirements on read and write operations

A *read operation* is an operation that reads data into a mutable buffer sequence argument of a type meeting **MutableBufferSequence** requirements. The mutable buffer sequence specifies memory where the data should be placed. A read operation shall always fill a buffer in the sequence completely before proceeding to the next.

A *write operation* is an operation that writes data from a constant buffer sequence argument of a type meeting **ConstBufferSequence** requirements. The constant buffer sequence specifies memory where the data to be written is located. A write operation shall always write a buffer in the sequence completely before proceeding to the next.

If a read or write operation is also an **asynchronous operation**, the operation shall maintain one or more copies of the buffer sequence until such time as the operation no longer requires access to the memory specified by the buffers in the sequence. The program shall ensure the memory remains valid until:

- the last copy of the buffer sequence is destroyed, or
 - the completion handler for the asynchronous operation is invoked,
- whichever comes first.

5.3 Requirements on synchronous socket operations

In this section, *synchronous socket operations* are those member functions specified as two overloads, with and without an argument of type `error_code&`:

```
R f(A1 a1, A2 a2, ..., AN aN);
R f(A1 a1, A2 a2, ..., AN aN, error_code& ec);
```

For an object *s*, the conditions under which its synchronous socket operations may block the calling thread (C++Std [defns.block]) are determined as follows.

If:

- *s.non_blocking()* == true,

- the synchronous socket operation is specified in terms of a *POSIX* function other than `poll()`,
 - that *POSIX* function lists `EWOULDBLOCK` or `EAGAIN` in its failure conditions, and
 - the effects of the operation cannot be established immediately
- then the synchronous socket operation shall not block the calling thread. [Note: And the effects of the operation are not established.
—end note]
- Otherwise, the synchronous socket operation shall block the calling thread until the effects are established.

5.4 Requirements on asynchronous socket operations

In this library, *asynchronous socket operations* are those member functions having prefix `async_`.

For an object `s`, a program may initiate asynchronous socket operations such that there are multiple simultaneously outstanding asynchronous operations.

When there are multiple outstanding asynchronous *read operations* on `s`:

- having no argument `flags` of type `socket_base::message_flags`, or
- having an argument `flags` of type `socket_base::message_flags` but where `(flags & socket_base::message_out_of_band) == 0`

then the `buffers` are filled in the order in which these operations were issued. The order of invocation of the completion handlers for these operations is unspecified.

When there are multiple outstanding asynchronous *read operations* on `s` having an argument `flags` of type `socket_base::message_flags` where `(flags & socket_base::message_out_of_band) != 0` then the `buffers` are filled in the order in which these operations were issued.

When there are multiple outstanding asynchronous *write operations* on `s`, the `buffers` are transmitted in the order in which these operations were issued. The order of invocation of the completion handlers for these operations is unspecified.

5.5 Acceptable protocol requirements

A type `X` meets the `AcceptableProtocol` requirements if it satisfies the requirements of `Protocol` as well as the additional requirements listed below.

Table 1: `AcceptableProtocol` requirements

expression	return type	assertion/note pre/post-conditions
<code>X::socket</code>	A type that satisfies the requirements of <code>Destructible</code> (C++Std [destructible]) and <code>MoveConstructible</code> (C++Std [moveconstructible]), and that is publicly and unambiguously derived from <code>basic_socket<X></code> .	

5.6 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 lambda as an accept handler:

```
acceptor.async_accept(...,
    [](const asio::error_code& ec)
    {
        ...
    });
});
```

A non-static class member function adapted to an accept handler using `std::bind()`:

```
void my_class::accept_handler(
    const asio::error_code& ec)
{
    ...
}
...
acceptor.async_accept(...,
    std::bind(&my_class::accept_handler,
        this, std::placeholders::_1));
```

A non-static class member function adapted to an accept handler using `boost::bind()`:

```
void my_class::accept_handler(
    const asio::error_code& ec)
{
    ...
}
...
acceptor.async_accept(...,
    boost::bind(&my_class::accept_handler,
        this, asio::placeholders::error));
```

5.7 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 2: Buffer-oriented asynchronous random-access read device requirements

operation	type	semantics, pre/post-conditions
<code>a.get_executor()</code>	A type satisfying the Executor requirements .	Returns the associated I/O executor.
<code>a.async_read_some_at(o, mb, h);</code>	<code>void</code>	<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>

5.8 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 3: Buffer-oriented asynchronous random-access write device requirements

operation	type	semantics, pre/post-conditions
<code>a.get_executor()</code>	A type satisfying the Executor requirements .	Returns the associated I/O executor.
<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>

5.9 Buffer-oriented asynchronous read stream requirements

A type X meets the `AsyncReadStream` requirements if it satisfies the requirements listed below.

In the table below, a denotes a value of type X, mb denotes a (possibly const) value satisfying the `MutableBufferSequence` requirements, and t is a completion token.

Table 4: `AsyncReadStream` requirements

operation	type	semantics, pre/post-conditions
<code>a.get_executor()</code>	A type satisfying the Executor requirements .	Returns the associated I/O executor.
<code>a.async_read_some(mb, t)</code>	The return type is determined according to the requirements for an asynchronous operation .	<p>Meets the requirements for a read operation and an asynchronous operation with completion signature <code>void(error_code ec, size_t n)</code>.</p> <p>If <code>buffer_size(mb) > 0</code>, initiates an asynchronous operation to read one or more bytes of data from the stream a into the buffer sequence mb. If successful, ec is set such that !ec is true, and n is the number of bytes read. If an error occurred, ec is set such that !ec is true, and n is 0. If all data has been read from the stream, and the stream performed an orderly shutdown, ec is <code>stream_errc::eof</code> and n is 0.</p> <p>If <code>buffer_size(mb) == 0</code>, the operation completes immediately. ec is set such that !ec is true, and n is 0.</p>

5.10 Buffer-oriented asynchronous write stream requirements

A type X meets the `AsyncWriteStream` requirements if it satisfies the requirements listed below.

In the table below, a denotes a value of type X, cb denotes a (possibly const) value satisfying the `ConstBufferSequence` requirements, and t is a completion token.

Table 5: `AsyncWriteStream` requirements

operation	type	semantics, pre/post-conditions
<code>a.get_executor()</code>	A type satisfying the Executor requirements .	Returns the associated I/O executor.

Table 5: (continued)

operation	type	semantics, pre/post-conditions
a.async_write_some(cb, t)	The return type is determined according to the requirements for an asynchronous operation .	<p>Meets the requirements for a write operation and an asynchronous operation with completion signature <code>void(error_code ec, size_t n)</code>.</p> <p>If <code>buffer_size(cb) > 0</code>, initiates an asynchronous operation to write one or more bytes of data to the stream <code>a</code> from the buffer sequence <code>cb</code>. If successful, <code>ec</code> is set such that <code>!ec</code> is <code>true</code>, and <code>n</code> is the number of bytes written. If an error occurred, <code>ec</code> is set such that <code>!ec</code> is <code>true</code>, and <code>n</code> is 0.</p> <p>If <code>buffer_size(cb) == 0</code>, the operation completes immediately. <code>ec</code> is set such that <code>!ec</code> is <code>true</code>, and <code>n</code> is 0.</p>

5.11 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 `boost::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));
```

5.12 Completion condition requirements

A *completion condition* is a function object that is used with the algorithms `read`, `async_read`, `write`, and `async_write` to determine when the algorithm has completed transferring data.

A type `X` meets the `CompletionCondition` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `CopyConstructible` (C++Std [copyconstructible]), as well as the additional requirements listed below.

In the table below, `x` denotes a value of type `X`, `ec` denotes a (possibly const) value of type `error_code`, and `n` denotes a (possibly const) value of type `size_t`.

Table 6: `CompletionCondition` requirements

expression	return type	assertion/note pre/post-condition
<code>x(ec, n)</code>	<code>size_t</code>	Let <code>n</code> be the total number of bytes transferred by the read or write algorithm so far. Returns the maximum number of bytes to be transferred on the next <code>read_some</code> , <code>async_read_some</code> , <code>write_some</code> , or <code>async_write_some</code> operation performed by the algorithm. Returns 0 to indicate that the algorithm is complete.

5.13 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 lambda as a completion handler:

```
my_io_service.post(
    []()
    {
        ...
    });

```

A non-static class member function adapted to a completion handler using `std::bind()`:

```
void my_class::completion_handler()
{
    ...
}
...
my_io_service.post(std::bind(&my_class::completion_handler, this));
```

A non-static class member function adapted to a completion handler using `boost::bind()`:

```
void my_class::completion_handler()
{
    ...
}
...
my_io_service.post(boost::bind(&my_class::completion_handler, this));
```

5.14 Connect condition requirements

A type `X` meets the `ConnectCondition` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `CopyConstructible` (C++Std [copyconstructible]), as well as the additional requirements listed below.

In the table below, `x` denotes a value of type `X`, `ec` denotes a (possibly const) value of type `error_code`, and `ep` denotes a (possibly const) value of some type satisfying the `endpoint` requirements.

Table 7: ConnectCondition requirements

expression	return type	assertion/note pre/post-condition
x(ec, ep)	bool	Returns <code>true</code> to indicate that the <code>connect</code> or <code>async_connect</code> algorithm should attempt a connection to the endpoint <code>ep</code> . Otherwise, returns <code>false</code> to indicate that the algorithm should not attempt connection to the endpoint <code>ep</code> , and should instead skip to the next endpoint in the sequence.

5.15 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 lambda as a connect handler:

```
socket.async_connect(...,
    [](const asio::error_code& ec)
    {
        ...
    });
});
```

A non-static class member function adapted to a connect handler using `std::bind()`:

```
void my_class::connect_handler(
    const asio::error_code& ec)
{
```

```

    ...
}

...
socket.async_connect(...,
    std::bind(&my_class::connect_handler,
        this, std::placeholders::_1));

```

A non-static class member function adapted to a connect handler using `boost::bind()`:

```

void my_class::connect_handler(
    const asio::error_code& ec)
{
    ...
}
...
socket.async_connect(...,
    boost::bind(&my_class::connect_handler,
        this, asio::placeholders::error));

```

5.16 Constant buffer sequence requirements

A *constant buffer sequence* represents a set of memory regions that may be used as input to an operation, such as the `send` operation of a socket.

A type `X` meets the `ConstBufferSequence` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `CopyConstructible` (C++Std [copyconstructible]), as well as the additional requirements listed below.

In the table below, `x` denotes a (possibly `const`) value of type `X`, and `u` denotes an identifier.

Table 8: ConstBufferSequence requirements

expression	return type	assertion/note pre/post-condition
<code>asio::buffer_sequence_begin(x)</code> <code>asio::buffer_sequence_end(x)</code>	An iterator type meeting the requirements for bidirectional iterators (C++Std [bidirectional.iterators]) whose value type is convertible to <code>const_buffer</code> .	

Table 8: (continued)

expression	return type	assertion/note pre/post-condition
X u(x);		<p>post:</p> <pre> equal(asio:: ↔ buffer_sequence_begin(x), asio:: ↔ buffer_sequence_end(x), asio:: ↔ buffer_sequence_begin(u), asio:: ↔ buffer_sequence_end(u), [] (const const_buffer& ← b1, const const_buffer& ← b2) { return b1.data() == ← b2.data() && b1.size() == ← b2.size(); }) </pre>

5.17 Dynamic buffer requirements

A dynamic buffer encapsulates memory storage that may be automatically resized as required, where the memory is divided into an input sequence followed by an output sequence. These memory regions are internal to the dynamic buffer sequence, but direct access to the elements is provided to permit them to be efficiently used with I/O operations, such as the `send` or `receive` operations of a socket. Data written to the output sequence of a dynamic buffer sequence object is appended to the input sequence of the same object.

A dynamic buffer type X shall satisfy the requirements of `MoveConstructible` (C++ Std, [moveconstructible]) types in addition to those listed below.

In the table below, X denotes a dynamic buffer class, x denotes a value of type `X&`, x1 denotes values of type `const X&`, and n denotes a value of type `size_t`, and u denotes an identifier.

Table 9: DynamicBuffer requirements

expression	type	assertion/note pre/post-conditions
<code>X::const_buffers_type</code>	type meeting ConstBufferSequence requirements.	This type represents the memory associated with the input sequence.
<code>X::mutable_buffers_type</code>	type meeting MutableBufferSequence requirements.	This type represents the memory associated with the output sequence.
<code>x1.size()</code>	<code>size_t</code>	Returns the size, in bytes, of the input sequence.

Table 9: (continued)

expression	type	assertion/note pre/post-conditions
<code>x1.max_size()</code>	<code>size_t</code>	Returns the permitted maximum of the sum of the sizes of the input sequence and output sequence.
<code>x1.capacity()</code>	<code>size_t</code>	Returns the maximum sum of the sizes of the input sequence and output sequence that the dynamic buffer can hold without requiring reallocation.
<code>x1.data()</code>	<code>X::const_buffers_type</code>	Returns a constant buffer sequence <code>u</code> that represents the memory associated with the input sequence, and where <code>buffer_size(u) == size()</code> .
<code>x.prepare(n)</code>	<code>X::mutable_buffers_type</code>	<p>Requires: <code>size() + n <= max_size()</code>.</p> <p>Returns a mutable buffer sequence <code>u</code> representing the output sequence, and where <code>buffer_size(u) == n</code>. The dynamic buffer reallocates memory as required. All constant or mutable buffer sequences previously obtained using <code>data()</code> or <code>prepare()</code> are invalidated.</p> <p>Throws: <code>length_error</code> if <code>size() + n > max_size()</code>.</p>
<code>x.commit(n)</code>		Appends <code>n</code> bytes from the start of the output sequence to the end of the input sequence. The remainder of the output sequence is discarded. If <code>n</code> is greater than the size of the output sequence, the entire output sequence is appended to the input sequence. All constant or mutable buffer sequences previously obtained using <code>data()</code> or <code>prepare()</code> are invalidated.
<code>x.consume(n)</code>		Removes <code>n</code> bytes from beginning of the input sequence. If <code>n</code> is greater than the size of the input sequence, the entire input sequence is removed. All constant or mutable buffer sequences previously obtained using <code>data()</code> or <code>prepare()</code> are invalidated.

5.18 Endpoint requirements

A type X meets the Endpoint requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]), `DefaultConstructible` (C++Std [defaultconstructible]), `CopyConstructible` (C++Std [copyconstructible]), and `CopyAssignable` (C++Std [copyassignable]), as well as the additional requirements listed below.

In the table below, a denotes a (possibly const) value of type X, and u denotes an identifier.

Table 10: Endpoint requirements

expression	type	assertion/note pre/post-conditions
<code>X:::protocol_type</code>	type meeting <code>Protocol</code> requirements	
<code>a.protocol()</code>	<code>protocol_type</code>	

In the table below, a denotes a (possibly const) value of type X, b denotes a value of type X, and s denotes a (possibly const) value of a type that is convertible to `size_t` and denotes a size in bytes.

Table 11: Endpoint requirements for extensible implementations

expression	type	assertion/note pre/post-conditions
<code>a.data()</code>	<code>const void*</code>	Returns a pointer suitable for passing as the <code>address</code> argument to functions such as <code>POSIX connect()</code> , or as the <code>dest_addr</code> argument to functions such as <code>POSIX sendto()</code> . The implementation shall perform a <code>static_cast</code> on the pointer to convert it to <code>const sockaddr*</code> .
<code>b.data()</code>	<code>void*</code>	Returns a pointer suitable for passing as the <code>address</code> argument to functions such as <code>POSIX accept()</code> , <code>getpeername()</code> , <code>getsockname()</code> and <code>recvfrom()</code> . The implementation shall perform a <code>static_cast</code> on the pointer to convert it to <code>sockaddr*</code> .
<code>a.size()</code>	<code>size_t</code>	Returns a value suitable for passing as the <code>address_len</code> argument to functions such as <code>POSIX connect()</code> , or as the <code>dest_len</code> argument to functions such as <code>POSIX sendto()</code> , after appropriate integer conversion has been performed.

Table 11: (continued)

expression	type	assertion/note pre/post-conditions
b.resize(s)		pre: $s \geq 0$ post: $a.size() == s$ Passed the value contained in the <i>address_len</i> argument to functions such as <i>POSIX accept()</i> , <i>getpeername()</i> , <i>getsockname()</i> and <i>recvfrom()</i> , after successful completion of the function. Permitted to throw an exception if the protocol associated with the endpoint object <i>a</i> does not support the specified size.
a.capacity()	size_t	Returns a value suitable for passing as the <i>address_len</i> argument to functions such as <i>POSIX accept()</i> , <i>getpeername()</i> , <i>getsockname()</i> and <i>recvfrom()</i> , after appropriate integer conversion has been performed.

5.19 Endpoint sequence requirements

A type *X* meets the `EndpointSequence` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `CopyConstructible` (C++Std [copyconstructible]), as well as the additional requirements listed below.

In the table below, *x* denotes a (possibly const) value of type *X*.

Table 12: `EndpointSequence` requirements

expression	return type	assertion/note pre/post-condition
<i>x.begin()</i> <i>x.end()</i>	A type meeting the requirements for forward iterators (C++Std [forward.iterators]) whose value type is convertible to a type satisfying the <code>Endpoint</code> requirements.	[<i>x.begin()</i> , <i>x.end()</i>) is a valid range.

5.20 Execution context requirements

A type *X* meets the `ExecutionContext` requirements if it is publicly and unambiguously derived from `execution_context`, and satisfies the additional requirements listed below.

In the table below, x denotes a value of type X .

Table 13: ExecutionContext requirements

expression	return type	assertion/note pre/post-condition
$X::executor_type$	type meeting Executor requirements	
$x.\sim X()$		Destroys all unexecuted function objects that were submitted via an executor object that is associated with the execution context.
$x.get_executor()$	$X::executor_type$	Returns an executor object that is associated with the execution context.

5.21 Executor requirements

The library describes a standard set of requirements for *executors*. A type meeting the **Executor** requirements embodies a set of rules for determining how submitted function objects are to be executed.

A type X meets the **Executor** requirements if it satisfies the requirements of **CopyConstructible** (C++Std [copyconstructible]) and **Destructible** (C++Std [destructible]), as well as the additional requirements listed below.

No constructor, comparison operator, copy operation, move operation, swap operation, or member functions `context`, `on_work_started`, and `on_work_finished` on these types shall exit via an exception.

The executor copy constructor, comparison operators, and other member functions defined in these requirements shall not introduce data races as a result of concurrent calls to those functions from different threads.

Let ctx be the execution context returned by the executor's `context()` member function. An executor becomes *invalid* when the first call to `ctx.shutdown()` returns. The effect of calling `on_work_started`, `on_work_finished`, `dispatch`, `post`, or `defer` on an invalid executor is undefined. [Note: The copy constructor, comparison operators, and `context()` member function continue to remain valid until ctx is destroyed. —end note]

In the table below, x_1 and x_2 denote (possibly const) values of type X , mx_1 denotes an x value of type X , f denotes a **MoveConstructible** (C++Std [moveconstructible]) function object callable with zero arguments, a denotes a (possibly const) value of type A meeting the **Allocator** requirements (C++Std [allocator.requirements]), and u denotes an identifier.

Table 14: Executor requirements

expression	type	assertion/note pre/post-conditions
$X u(x_1);$		<p>Shall not exit via an exception.</p> <p>post: $u == x_1$ and $\text{std}::addressof(u.context()) == \text{std}::addressof(x_1.context())$.</p>

Table 14: (continued)

expression	type	assertion/note pre/post-conditions
<code>x_u(mx1);</code>		Shall not exit via an exception. post: u equals the prior value of mx1 and std::addressof(u.context()) equals the prior value of std::addressof(mx1.context()).
<code>x1 == x2</code>	bool	Returns <code>true</code> only if x1 and x2 can be interchanged with identical effects in any of the expressions defined in these type requirements. [Note: Returning <code>false</code> does not necessarily imply that the effects are not identical. —end note] <code>operator==</code> shall be reflexive, symmetric, and transitive, and shall not exit via an exception.
<code>x1 != x2</code>	bool	Same as <code>!(x1 == x2)</code> .
<code>x1.context()</code>	<code>execution_context&</code> , or <code>E&</code> where E is a type that satisfies the <code>ExecutionContext</code> requirements.	Shall not exit via an exception. The comparison operators and member functions defined in these requirements shall not alter the reference returned by this function.
<code>x1.on_work_started()</code>		Shall not exit via an exception.
<code>x1.on_work_finished()</code>		Shall not exit via an exception. Precondition: A preceding call <code>x2.on_work_started()</code> where <code>x1 == x2</code> .

Table 14: (continued)

expression	type	assertion/note pre/post-conditions
x1.dispatch(std::move(f), a)		<p>Effects: Creates an object <code>f1</code> initialized with <code>DECAY_COPY(forward<Func>(f))</code> (<code>C++Std [thread.decaycopy]</code>) in the current thread of execution. Calls <code>f1()</code> at most once. The executor may block forward progress of the caller until <code>f1()</code> finishes execution.</p> <p>Executor implementations should use the supplied allocator to allocate any memory required to store the function object. Prior to invoking the function object, the executor shall deallocate any memory allocated. <i>[Note: Executors defined in this Technical Specification always use the supplied allocator unless otherwise specified. —end note]</i></p> <p>Synchronization: The invocation of <code>dispatch</code> synchronizes with (<code>C++Std [intro.multithread]</code>) the invocation of <code>f1</code>.</p>

Table 14: (continued)

expression	type	assertion/note pre/post-conditions
<pre>x1.post(std::move(f), a) x1.defer(std::move(f), a)</pre>		<p>Effects: Creates an object <code>f1</code> initialized with <code>DECAY_COPY(forward<Func>(f))</code> in the current thread of execution. Calls <code>f1()</code> at most once. The executor shall not block forward progress of the caller pending completion of <code>f1()</code>.</p> <p>Executor implementations should use the supplied allocator to allocate any memory required to store the function object. Prior to invoking the function object, the executor shall deallocate any memory allocated. [Note: Executors defined in this Technical Specification always use the supplied allocator unless otherwise specified. —end note]</p> <p>Synchronization: The invocation of <code>post</code> or <code>defer</code> synchronizes with (C++Std [intro.multithread]) the invocation of <code>f1</code>.</p> <p>[Note: Although the requirements placed on <code>defer</code> are identical to <code>post</code>, the use of <code>post</code> conveys a preference that the caller <i>does not</i> block the first step of <code>f1</code>'s progress, whereas <code>defer</code> conveys a preference that the caller <i>does</i> block the first step of <code>f1</code>. One use of <code>defer</code> is to convey the intention of the caller that <code>f1</code> is a continuation of the current call context. The executor may use this information to optimize or otherwise adjust the way in which <code>f1</code> is invoked. —end note]</p>

5.22 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 15: GettableSerialPortOption requirements

expression	type	assertion/note pre/post-conditions
<code>const storage& u = s; a.load(u, ec);</code>	<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>

5.23 Gettable socket option requirements

A type `X` meets the `GettableSocketOption` requirements if it satisfies the requirements listed below.

In the table below, `a` denotes a (possibly const) value of type `X`, `b` denotes a value of type `X`, `p` denotes a (possibly const) value that meets the [Protocol](#) requirements, and `s` denotes a (possibly const) value of a type that is convertible to `size_t` and denotes a size in bytes.

Table 16: GettableSocketOption requirements for extensible implementations

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 <code>POSIX getsockopt()</code> (or equivalent).
<code>a.name(p)</code>	<code>int</code>	Returns a value suitable for passing as the <code>option_name</code> argument to <code>POSIX getsockopt()</code> (or equivalent).
<code>b.data(p)</code>	<code>void*</code>	Returns a pointer suitable for passing as the <code>option_value</code> argument to <code>POSIX getsockopt()</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 getsockopt()</code> (or equivalent), after appropriate integer conversion has been performed.
<code>b.resize(p, s)</code>		<p>post: <code>b.size(p) == s</code>.</p> <p>Passed the value contained in the <code>option_len</code> argument to <code>POSIX getsockopt()</code> (or equivalent) after successful completion of the function.</p> <p>Permitted to throw an exception if the socket option object <code>b</code> does not support the specified size.</p>

5.24 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 17: 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 s. 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 p, of at least size s, 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 f 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>

5.25 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 lambda as a handshake handler:

```
ssl_stream.async_handshake(...,
    [](const asio::error_code& ec)
    {
        ...
    });
});
```

A non-static class member function adapted to a handshake handler using `std::bind()`:

```
void my_class::handshake_handler(
    const asio::error_code& ec)
{
    ...
}
...
ssl_stream.async_handshake(...,
    std::bind(&my_class::handshake_handler,
        this, std::placeholders::_1));
```

A non-static class member function adapted to a handshake handler using `boost::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));
```

5.26 Internet protocol requirements

A type X meets the `InternetProtocol` requirements if it satisfies the requirements of `AcceptableProtocol`, as well as the additional requirements listed below.

In the table below, a denotes a (possibly const) value of type X, and b denotes a (possibly const) value of type X.

Table 18: InternetProtocol requirements

expression	return type	assertion/note pre/post-conditions
<code>X::resolver</code>	<code>ip::basic_resolver<X></code>	The type of a resolver for the protocol.
<code>X::v4()</code>	<code>X</code>	Returns an object representing the IP version 4 protocol.
<code>X::v6()</code>	<code>X</code>	Returns an object representing the IP version 6 protocol.
<code>a == b</code>	convertible to <code>bool</code>	Returns <code>true</code> if a and b represent the same IP protocol version, otherwise <code>false</code> .
<code>a != b</code>	convertible to <code>bool</code>	Returns <code>!(a == b)</code> .

5.27 I/O control command requirements

A type X meets the `IoControlCommand` requirements if it satisfies the requirements listed below.

In the table below, a denotes a (possibly const) value of type X, and b denotes a value of type X.

Table 19: IoControlCommand requirements for extensible implementations

expression	type	assertion/note pre/post-conditions
<code>a.name()</code>	<code>int</code>	Returns a value suitable for passing as the <code>request</code> argument to <i>POSIX ioctl()</i> (or equivalent).
<code>b.data()</code>	<code>void*</code>	

5.28 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 20: IoObjectService requirements

expression	return type	assertion/note pre/post-condition
<code>X::implementation_type</code>		
<code>X::implementation_type u;</code>		note: <code>X::implementation_type</code> has a public default constructor and destructor.
<code>a.construct(b);</code>		
<code>a.destroy(b);</code>		note: <code>destroy()</code> will only be called on a value that has previously been initialised with <code>construct()</code> or <code>move_construct()</code> .
<code>a.move_construct(b, c);</code>		note: only required for I/O objects that support movability.
<code>a.move_assign(b, ao, c);</code>		note: only required for I/O objects that support movability.

5.29 Iterator connect handler requirements

An iterator connect handler must meet the requirements for a [handler](#). A value `h` of an iterator 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 an iterator connect handler:

```
void connect_handler(
    const asio::error_code& ec,
    asio::ip::tcp::resolver::iterator iterator)
{
    ...
}
```

An iterator connect handler function object:

```
struct connect_handler
{
    ...
    template <typename Iterator>
    void operator()(

        const asio::error_code& ec,
        Iterator iterator)
    {
    }
}
```

```
    ...
}
```

A lambda as an iterator connect handler:

```
asio::async_connect(...,
    [](const asio::error_code& ec,
       asio::ip::tcp::resolver::iterator iterator)
{
    ...
});
```

A non-static class member function adapted to an iterator connect handler using `std::bind()`:

```
void my_class::connect_handler(
    const asio::error_code& ec,
    asio::ip::tcp::resolver::iterator iterator)
{
    ...
}
...
asio::async_connect(...,
    std::bind(&my_class::connect_handler,
              this, std::placeholders::_1,
              std::placeholders::_2));
```

A non-static class member function adapted to an iterator connect handler using `boost::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));
```

5.30 Move accept handler requirements

A move accept handler must meet the requirements for a [handler](#). A value `h` of a move accept 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 the nested type `Protocol::socket` for the type `Protocol` of the socket class template.

Examples

A free function as a move accept handler:

```
void accept_handler(
    const asio::error_code& ec, asio::ip::tcp::socket s)
{
    ...
}
```

A move accept handler function object:

```

struct accept_handler
{
    ...
    void operator()(const asio::error_code& ec, asio::ip::tcp::socket s)
    {
        ...
    }
    ...
};

};


```

A lambda as a move accept handler:

```

acceptor.async_accept(...,
    [](const asio::error_code& ec, asio::ip::tcp::socket s)
    {
        ...
    });

```

A non-static class member function adapted to a move accept handler using `std::bind()`:

```

void my_class::accept_handler(
    const asio::error_code& ec, asio::ip::tcp::socket socket)
{
    ...
}
...
asio::async_accept(...,
    std::bind(&my_class::accept_handler,
        this, std::placeholders::_1,
        std::placeholders::_2));

```

5.31 Mutable buffer sequence requirements

A *mutable buffer sequence* represents a set of memory regions that may be used to receive the output of an operation, such as the `receive` operation of a socket.

A type `X` meets the `MutableBufferSequence` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `CopyConstructible` (C++Std [copyconstructible]), as well as the additional requirements listed below.

In the table below, `x` denotes a (possibly const) value of type `X`, and `u` denotes an identifier.

Table 21: `MutableBufferSequence` requirements

expression	return type	assertion/note pre/post-condition
<code>asio::buffer_sequence_begin(x)</code> <code>asio::buffer_sequence_end(x)</code>	An iterator type meeting the requirements for bidirectional iterators (C++Std [bidirectional.iterators]) whose value type is convertible to <code>mutable_buffer</code> .	

Table 21: (continued)

expression	return type	assertion/note pre/post-condition
X::u(x);		<p>post:</p> <pre> equal(asio:: ↔ buffer_sequence_begin(x), asio:: ↔ buffer_sequence_end(x), asio:: ↔ buffer_sequence_begin(u), asio:: ↔ buffer_sequence_end(u), [] (const mutable_buffer& ↔ b1, const mutable_buffer& ↔ b2) { return b1.data() == ↔ b2.data() && b1.size() == ↔ b2.size(); }) </pre>

5.32 Proto-allocator requirements

A type A meets the proto-allocator requirements if A is CopyConstructible (C++Std [copyconstructible]), Destructible (C++Std [destructible]), and allocator_traits<A>::rebind_alloc<U> meets the allocator requirements (C++Std [allocator.requirements]), where U is an object type. [Note: For example, std::allocator<void> meets the proto-allocator requirements but not the allocator requirements. —end note] No constructor, comparison operator, copy operation, move operation, or swap operation on these types shall exit via an exception.

5.33 Protocol requirements

A type X meets the Protocol requirements if it satisfies the requirements of Destructible (C++Std [destructible]), CopyConstructible (C++Std [copyconstructible]), and CopyAssignable (C++Std [copyassignable]), as well as the additional requirements listed below.

Table 22: Protocol requirements

expression	return type	assertion/note pre/post-conditions
X::endpoint	type meeting endpoint requirements	

In the table below, a denotes a (possibly const) value of type X.

Table 23: Protocol requirements for extensible implementations

expression	return type	assertion/note pre/post-conditions
a.family()	int	Returns a value suitable for passing as the <i>domain</i> argument to <i>POSIX socket ()</i> (or equivalent).
a.type()	int	Returns a value suitable for passing as the <i>type</i> argument to <i>POSIX socket ()</i> (or equivalent).
a.protocol()	int	Returns a value suitable for passing as the <i>protocol</i> argument to <i>POSIX socket ()</i> (or equivalent).

5.34 Range connect handler requirements

A range connect handler must meet the requirements for a **handler**. A value *h* of a range connect handler class should work correctly in the expression *h(ec, ep)*, where *ec* is an lvalue of type `const error_code` and *ep* is an lvalue of the type `Protocol::endpoint` for the `Protocol` type in the corresponding `connect()` or `async_connect()` function.

Examples

A free function as a range connect handler:

```
void connect_handler(
    const asio::error_code& ec,
    const asio::ip::tcp::endpoint& endpoint)
{
    ...
}
```

A range connect handler function object:

```
struct connect_handler
{
    ...
    template <typename Range>
    void operator()(const asio::error_code& ec,
                    const asio::ip::tcp::endpoint& endpoint)
    {
        ...
    }
    ...
};
```

A lambda as a range connect handler:

```
asio::async_connect(...,
    [] (const asio::error_code& ec,
        const asio::ip::tcp::endpoint& endpoint)
```

```
{  
    ...  
});
```

A non-static class member function adapted to a range connect handler using `std::bind()`:

```
void my_class::connect_handler(  
    const asio::error_code& ec,  
    const asio::ip::tcp::endpoint& endpoint)  
{  
    ...  
}  
...  
asio::async_connect(...,  
    std::bind(&my_class::connect_handler,  
        this, std::placeholders::_1,  
        std::placeholders::_2));
```

A non-static class member function adapted to a range connect handler using `boost::bind()`:

```
void my_class::connect_handler(  
    const asio::error_code& ec,  
    const asio::ip::tcp::endpoint& endpoint)  
{  
    ...  
}  
...  
asio::async_connect(...,  
    boost::bind(&my_class::connect_handler,  
        this, asio::placeholders::error,  
        asio::placeholders::endpoint));
```

5.35 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 lambda as a read handler:

```
socket.async_read(...  
    [](const asio::error_code& ec,  
        std::size_t bytes_transferred)  
{  
    ...  
});
```

A non-static class member function adapted to a read handler using `std::bind()`:

```
void my_class::read_handler(  
    const asio::error_code& ec,  
    std::size_t bytes_transferred)  
{  
    ...  
}  
...  
socket.async_read(...,  
    std::bind(&my_class::read_handler,  
        this, std::placeholders::_1,  
        std::placeholders::_2));
```

A non-static class member function adapted to a read handler using `boost::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));
```

5.36 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, r)`, where `ec` is an lvalue of type `const error_code` and `r` is an lvalue of type `const ip::basic_resolver_result<InternetProtocol>`. `InternetProtocol` is the template parameter of the [`ip::basic_resolver<>`](#) 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::results_type results)  
{  
    ...  
}
```

A resolve handler function object:

```
struct resolve_handler  
{  
    ...  
}
```

```

void operator()(
    const asio::error_code& ec,
    asio::ip::tcp::resolver::results_type results)
{
    ...
}
...
};


```

A lambda as a resolve handler:

```

resolver.async_resolve(...,
    [](const asio::error_code& ec,
        asio::ip::tcp::resolver::results_type results)
    {
        ...
    });

```

A non-static class member function adapted to a resolve handler using `std::bind()`:

```

void my_class::resolve_handler(
    const asio::error_code& ec,
    asio::ip::tcp::resolver::results_type results)
{
    ...
}
...
resolver.async_resolve(...,
    std::bind(&my_class::resolve_handler,
        this, std::placeholders::_1,
        std::placeholders::_2));

```

A non-static class member function adapted to a resolve handler using `boost::bind()`:

```

void my_class::resolve_handler(
    const asio::error_code& ec,
    asio::ip::tcp::resolver::results_type results)
{
    ...
}
...
resolver.async_resolve(...,
    boost::bind(&my_class::resolve_handler,
        this, asio::placeholders::error,
        asio::placeholders::results));

```

5.37 Service requirements

A class is a *service* if it is publicly and unambiguously derived from `execution_context::service`, or if it is publicly and unambiguously derived from another service. For a service `S`, `S::key_type` shall be valid and denote a type (C++Std [temp.deduct]), `is_base_of_v<typename S::key_type, S>` shall be `true`, and `S` shall satisfy the `Destructible` requirements (C++Std [destructible]).

The first parameter of all service constructors shall be an lvalue reference to `execution_context`. This parameter denotes the `execution_context` object that represents a set of services, of which the service object will be a member. [Note: These constructors may be called by the `make_service` function. —end note]

A service shall provide an explicit constructor with a single parameter of lvalue reference to `execution_context`. [Note: This constructor may be called by the `use_service` function. —end note]

```

class my_service : public execution_context::service
{
public:
    typedef my_service key_type;
    explicit my_service(execution_context& ctx);
    my_service(execution_context& ctx, int some_value);
private:
    virtual void shutdown() noexcept override;
    ...
};

```

A service's shutdown member function shall destroy all copies of user-defined function objects that are held by the service.

5.38 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 24: 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> .

5.39 Settable socket option requirements

A type X meets the `SettableSocketOption` requirements if it satisfies the requirements listed below.

In the table below, a denotes a (possibly const) value of type X, p denotes a (possibly const) value that meets the `Protocol` requirements, and u denotes an identifier.

Table 25: SettableSocketOption requirements for extensible implementations

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 25: (continued)

expression	type	assertion/note pre/post-conditions
a.data(p)	const void*	Returns a pointer suitable for passing as the <i>option_value</i> argument to <i>POSIX setsockopt()</i> (or equivalent).
a.size(p)	size_t	Returns a value suitable for passing as the <i>option_len</i> argument to <i>POSIX setsockopt()</i> (or equivalent), after appropriate integer conversion has been performed.

5.40 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 lambda as a shutdown handler:

```
ssl_stream.async_shutdown(...,
    [](const asio::error_code& ec)
    {
        ...
    });
});
```

A non-static class member function adapted to a shutdown handler using `std::bind()`:

```

void my_class::shutdown_handler(
    const asio::error_code& ec)
{
    ...
}
...
ssl_stream.async_shutdown(
    std::bind(&my_class::shutdown_handler,
        this, std::placeholders::_1));

```

A non-static class member function adapted to a shutdown handler using `boost::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));

```

5.41 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 lambda as a signal handler:

```

my_signal_set.async_wait(
    [](const asio::error_code& ec,
        int signal_number)
    {
        ...
    });

```

A non-static class member function adapted to a signal handler using `std::bind()`:

```
void my_class::signal_handler(
    const asio::error_code& ec,
    int signal_number)
{
    ...
}
...
my_signal_set.async_wait(
    std::bind(&my_class::signal_handler,
        this, std::placeholders::_1,
        std::placeholders::_2));
```

A non-static class member function adapted to a signal handler using `boost::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));
```

5.42 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 26: Buffer-oriented synchronous random-access read device requirements

operation	type	semantics, pre/post-conditions
<code>a.read_some_at(o, mb);</code>	<code>size_t</code>	Equivalent to: <code>error_code ec;</code> <code>size_t s = a.read_some_at(↔</code> <code>o, mb, ec);</code> <code>if (ec) throw system_error ↔</code> <code>(ec);</code> <code>return s;</code>

Table 26: (continued)

operation	type	semantics, pre/post-conditions
<code>a.read_some_at(o, mb, ec);</code>	<code>size_t</code>	<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>

5.43 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 27: Buffer-oriented synchronous random-access write device requirements

operation	type	semantics, pre/post-conditions
<code>a.write_some_at(o, cb);</code>	<code>size_t</code>	<p>Equivalent to:</p> <pre>error_code ec; size_t s = a.write_some(o, ← cb, ec); if (ec) throw system_error ← (ec); return s;</pre>

Table 27: (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>

5.44 Buffer-oriented synchronous read stream requirements

A type `X` meets the `SyncReadStream` requirements if it satisfies the requirements listed below.

In the table below, `a` denotes a value of type `X`, `mb` denotes a (possibly const) value satisfying the `MutableBufferSequence` requirements, and `ec` denotes an object of type `error_code`.

Table 28: SyncReadStream requirements

operation	type	semantics, pre/post-conditions
<code>a.read_some(mb)</code> <code>a.read_some(mb, ec)</code>	<code>size_t</code>	<p>Meets the requirements for a <code>read</code> operation.</p> <p>If <code>buffer_size(mb) > 0</code>, reads one or more bytes of data from the stream <code>a</code> into the buffer sequence <code>mb</code>. If successful, sets <code>ec</code> such that <code>!ec</code> is true, and returns the number of bytes read. If an error occurred, sets <code>ec</code> such that <code>!ec</code> is true, and returns 0. If all data has been read from the stream, and the stream performed an orderly shutdown, sets <code>ec</code> to <code>stream_errc::eof</code> and returns 0.</p> <p>If <code>buffer_size(mb) == 0</code>, the operation shall not block. Sets <code>ec</code> such that <code>!ec</code> is true, and returns 0.</p>

5.45 Buffer-oriented synchronous write stream requirements

A type `X` meets the `SyncWriteStream` requirements if it satisfies the requirements listed below.

In the table below, `a` denotes a value of type `X`, `cb` denotes a (possibly const) value satisfying the `ConstBufferSequence` requirements, and `ec` denotes an object of type `error_code`.

Table 29: SyncWriteStream requirements

operation	type	semantics, pre/post-conditions
<code>a.write_some(cb)</code> <code>a.write_some(cb, ec)</code>	<code>size_t</code>	<p>Meets the requirements for a <code>write</code> operation.</p> <p>If <code>buffer_size(cb) > 0</code>, writes one or more bytes of data to the stream <code>a</code> from the buffer sequence <code>cb</code>. If successful, sets <code>ec</code> such that <code>!ec</code> is true, and returns the number of bytes written. If an error occurred, sets <code>ec</code> such that <code>!ec</code> is false, and returns 0.</p> <p>If <code>buffer_size(cb) == 0</code>, the operation shall not block. Sets <code>ec</code> such that <code>!ec</code> is true, and returns 0.</p>

5.46 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 30: TimeTraits requirements

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

Table 30: (continued)

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

5.47 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 lambda as a wait handler:

```

socket.async_wait(...,
    [] (const asio::error_code& ec)
{
    ...
});
```

A non-static class member function adapted to a wait handler using `std::bind()`:

```

void my_class::wait_handler(
    const asio::error_code& ec)
{
    ...
}
...
socket.async_wait(...,
    std::bind(&my_class::wait_handler,
        this, std::placeholders::_1));
```

A non-static class member function adapted to a wait handler using `boost::bind()`:

```

void my_class::wait_handler(
    const asio::error_code& ec)
{
    ...
}
...
socket.async_wait(...,
    boost::bind(&my_class::wait_handler,
        this, asio::placeholders::error));
```

5.48 Wait traits requirements

The `basic_waitable_timer` template uses wait traits to allow programs to customize `wait` and `async_wait` behavior.

[*Note:* Possible uses of wait traits include:

- To enable timers based on non-realtime clocks.
- Determining how quickly wallclock-based timers respond to system time changes.
- Correcting for errors or rounding timeouts to boundaries.
- Preventing duration overflow. That is, a program may set a timer's expiry e to be `Clock::max()` (meaning never reached) or `Clock::min()` (meaning always in the past). As a result, computing the duration until timer expiry as $e - \text{Clock::now}()$ may cause overflow. —*end note*

For a type `Clock` meeting the `Clock` requirements (C++Std [time.clock.req]), a type `X` meets the `WaitTraits` requirements if it satisfies the requirements listed below.

In the table below, `t` denotes a (possibly const) value of type `Clock::time_point`; and `d` denotes a (possibly const) value of type `Clock::duration`.

Table 31: WaitTraits requirements

expression	return type	assertion/note pre/post-condition
X::to_wait_duration(d)	Clock::duration	Returns a Clock::duration value to be used in a wait or async_wait operation. [Note: The return value is typically representative of the duration d. —end note]
X::to_wait_duration(t)	Clock::duration	Returns a Clock::duration value to be used in a wait or async_wait operation. [Note: The return value is typically representative of the duration from Clock::now() until the time point t. —end note]

5.49 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 lambda as a write handler:

```
socket.async_write(...,
    [](const asio::error_code& ec,
       std::size_t bytes_transferred)
    {
        ...
    });
});
```

A non-static class member function adapted to a write handler using `std::bind()`:

```
void my_class::write_handler(
    const asio::error_code& ec,
    std::size_t bytes_transferred)
{
    ...
}
...
socket.async_write(...,
    std::bind(&my_class::write_handler,
        this, std::placeholders::_1,
        std::placeholders::_2));
```

A non-static class member function adapted to a write handler using `boost::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));
```

5.50 add_service

```
template<
    typename Service>
void add_service(
    execution_context & e,
    Service * svc);
```

This function is used to add a service to the `execution_context`.

Parameters

e The `execution_context` object that owns the service.

svc The service object. On success, ownership of the service object is transferred to the `execution_context`. When the `execution_context` object is destroyed, it will destroy the service object by performing:

```
delete static_cast<execution_context::service*>(svc)
```

Exceptions

asio::service_already_exists Thrown if a service of the given type is already present in the `execution_context`.

asio::invalid_service_owner Thrown if the service's owning `execution_context` is not the `execution_context` object specified by the `e` parameter.

Requirements

Header: asio/impl/execution_context.hpp

Convenience header: asio.hpp

5.51 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

5.52 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

5.53 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_context` 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_context` object. Handlers may subsequently be invoked through other objects (such as `io_context::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

5.53.1 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,
    ... );
```

5.53.2 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,
    ... );
```

5.54 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`

5.55 associated_allocator

Traits type used to obtain the allocator associated with an object.

```
template<
    typename T,
    typename Allocator = std::allocator<void>>
struct associated_allocator
```

Types

Name	Description
type	If T has a nested type allocator_type, T::allocator_type. Otherwise Allocator.

Member Functions

Name	Description
get	If T has a nested type allocator_type, returns t.get_allocator(). Otherwise returns a.

A program may specialise this traits type if the T template parameter in the specialisation is a user-defined type. The template parameter Allocator shall be a type meeting the Allocator requirements.

Specialisations shall meet the following requirements, where t is a const reference to an object of type T, and a is an object of type Allocator.

- Provide a nested typedef type that identifies a type meeting the Allocator requirements.
- Provide a noexcept static member function named get, callable as get (t) and with return type type.
- Provide a noexcept static member function named get, callable as get (t, a) and with return type type.

Requirements

Header: asio/associated_allocator.hpp

Convenience header: asio.hpp

5.55.1 associated_allocator::get

If T has a nested type allocator_type, returns t.get_allocator (). Otherwise returns a.

```
static type get(
    const T & t,
    const Allocator & a = Allocator());
```

5.55.2 associated_allocator::type

If T has a nested type allocator_type, T::allocator_type. Otherwise Allocator.

```
typedef see_below type;
```

Requirements

Header: asio/associated_allocator.hpp

Convenience header: asio.hpp

5.56 associated_executor

Traits type used to obtain the executor associated with an object.

```
template<
    typename T,
    typename Executor = system_executor>
struct associated_executor
```

Types

Name	Description
type	If T has a nested type executor_type, T::executor_type. Otherwise Executor.

Member Functions

Name	Description
get	If T has a nested type executor_type, returns t.get_executor(). Otherwise returns ex.

A program may specialise this traits type if the T template parameter in the specialisation is a user-defined type. The template parameter Executor shall be a type meeting the Executor requirements.

Specialisations shall meet the following requirements, where t is a const reference to an object of type T, and e is an object of type Executor.

- Provide a nested typedef type that identifies a type meeting the Executor requirements.
- Provide a noexcept static member function named get, callable as get (t) and with return type type.
- Provide a noexcept static member function named get, callable as get (t, e) and with return type type.

Requirements

Header: asio/associated_executor.hpp

Convenience header: asio.hpp

5.56.1 associated_executor::get

If T has a nested type executor_type, returns t.get_executor(). Otherwise returns ex.

```
static type get(
    const T & t,
    const Executor & ex = Executor());
```

5.56.2 associated_executor::type

If T has a nested type executor_type, T::executor_type. Otherwise Executor.

```
typedef see_below type;
```

Requirements

Header: asio/associated_executor.hpp

Convenience header: asio.hpp

5.57 async_completion

Helper template to deduce the handler type from a CompletionToken, capture a local copy of the handler, and then create an [async_result](#) for the handler.

```
template<
    typename CompletionToken,
    typename Signature>
struct async_completion
```

Types

Name	Description
completion_handler_type	The real handler type to be used for the asynchronous operation.

Member Functions

Name	Description
async_completion	Constructor.

Data Members

Name	Description
completion_handler	A copy of, or reference to, a real handler object.
result	The result of the asynchronous operation's initiating function.

Requirements

Header: asio/async_result.hpp

Convenience header: asio.hpp

5.57.1 `async_completion::async_completion`

Constructor.

```
async_completion(CompletionToken & token);
```

The constructor creates the concrete completion handler and makes the link between the handler and the asynchronous result.

5.57.2 `async_completion::completion_handler`

A copy of, or reference to, a real handler object.

```
conditional< is_same< CompletionToken, completion_handler_type >::value, ←
    completion_handler_type &, completion_handler_type >::type completion_handler;
```

5.57.3 `async_completion::completion_handler_type`

The real handler type to be used for the asynchronous operation.

```
typedef asio::async_result< typename decay< CompletionToken >::type, Signature >::←
    completion_handler_type completion_handler_type;
```

Types

Name	Description
completion_handler_type	The concrete completion handler type for the specific signature.
return_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.

The `async_result` traits class is used for determining:

- the concrete completion handler type to be called at the end of the asynchronous operation;
- the initiating function return type; and
- how the return value of the initiating function is obtained.

The trait allows the handler and return types to be determined at the point where the specific completion handler signature is known. This template may be specialised for user-defined completion token types. The primary template assumes that the `CompletionToken` is the completion handler.

Requirements

Header: asio/async_result.hpp

Convenience header: asio.hpp

5.57.4 `async_completion::result`

The result of the asynchronous operation's initiating function.

```
async_result< typename decay< CompletionToken >::type, Signature > result;
```

5.58 `async_connect`

Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename EndpointSequence,
    typename RangeConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    RangeConnectHandler && handler,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);

template<
    typename Protocol,
    typename Iterator,
    typename IteratorConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    IteratorConnectHandler && handler,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);

template<
    typename Protocol,
    typename Iterator,
    typename IteratorConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    Iterator end,
    IteratorConnectHandler && handler);

template<
    typename Protocol,
    typename EndpointSequence,
    typename ConnectCondition,
    typename RangeConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    ConnectCondition connect_condition,
    RangeConnectHandler && handler,
```

```

typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);

template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition,
    typename IteratorConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    ConnectCondition connect_condition,
    IteratorConnectHandler && handler,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);

template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition,
    typename IteratorConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    Iterator end,
    ConnectCondition connect_condition,
    IteratorConnectHandler && handler);

```

Requirements

Header: asio/connect.hpp

Convenience header: asio.hpp

5.58.1 `async_connect` (1 of 6 overloads)

Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```

template<
    typename Protocol,
    typename EndpointSequence,
    typename RangeConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    RangeConnectHandler && handler,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);

```

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.

endpoints 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, the successfully connected endpoint.
    // Otherwise, a default-constructed endpoint.
    const typename Protocol::endpoint& endpoint
);
```

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_context::post()`.

Example

```
tcp::resolver r(io_context);
tcp::resolver::query q("host", "service");
tcp::socket s(io_context);

// ...

r.async_resolve(q, resolve_handler);

// ...

void resolve_handler(
    const asio::error_code& ec,
    tcp::resolver::results_type results)
{
    if (!ec)
    {
        asio::async_connect(s, results, connect_handler);
    }
}

// ...

void connect_handler(
    const asio::error_code& ec,
    const tcp::endpoint& endpoint)
{
    // ...
}
```

5.58.2 `async_connect` (2 of 6 overloads)

(Deprecated.) Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename Iterator,
    typename IteratorConnectHandler>
DEDUCED async_connect(
    basic_socket<Protocol> & s,
    Iterator begin,
```

```
IteratorConnectHandler && handler,
typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);
```

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_context::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`.

5.58.3 `async_connect` (3 of 6 overloads)

Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename Iterator,
    typename IteratorConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    Iterator end,
    IteratorConnectHandler && 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_context::post()`.

Example

```
std::vector<tcp::endpoint> endpoints = ...;
tcp::socket s(io_context);
asio::async_connect(s,
    endpoints.begin(), endpoints.end(),
    connect_handler);

// ...

void connect_handler(
    const asio::error_code& ec,
    std::vector<tcp::endpoint>::iterator i)
{
    // ...
}
```

5.58.4 `async_connect` (4 of 6 overloads)

Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename EndpointSequence,
    typename ConnectCondition,
    typename RangeConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    ConnectCondition connect_condition,
    RangeConnectHandler && handler,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);
```

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.

endpoints 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:

```
bool connect_condition(
    const asio::error_code& ec,
    const typename Protocol::endpoint& 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 the next endpoint to be tried. The function object should return true if the next endpoint should be tried, and false if it should be skipped.

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_context::post()`.

Example

The following connect condition function object can be used to output information about the individual connection attempts:

```
struct my_connect_condition
{
    bool operator()(const asio::error_code& ec,
                     const ::tcp::endpoint& next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next << std::endl;
        return true;
    }
};
```

It would be used with the `asio::connect` function as follows:

```
tcp::resolver r(io_context);
tcp::resolver::query q("host", "service");
tcp::socket s(io_context);

// ...

r.async_resolve(q, resolve_handler);

// ...

void resolve_handler(const asio::error_code& ec,
                     tcp::resolver::results_type results)
{
    if (!ec)
```

```

    {
        asio::async_connect(s, results,
            my_connect_condition(),
            connect_handler);
    }
}

// ...

void connect_handler(
    const asio::error_code& ec,
    const tcp::endpoint& endpoint)
{
    if (ec)
    {
        // An error occurred.
    }
    else
    {
        std::cout << "Connected to: " << endpoint << std::endl;
    }
}

```

5.58.5 `async_connect` (5 of 6 overloads)

(Deprecated.) Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```

template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition,
    typename IteratorConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    ConnectCondition connect_condition,
    IteratorConnectHandler && handler,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);

```

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:

```

bool connect_condition(
    const asio::error_code& ec,
    const typename Protocol::endpoint& 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 the next endpoint to be tried. The function object should return true if the next endpoint should be tried, and false if it should be skipped.

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_context::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`.

5.58.6 `async_connect` (6 of 6 overloads)

Asynchronously establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition,
    typename IteratorConnectHandler>
DEDUCED async_connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    Iterator end,
    ConnectCondition connect_condition,
    IteratorConnectHandler && 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:

```
bool connect_condition(
    const asio::error_code& ec,
    const typename Protocol::endpoint& 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 the next endpoint to be tried. The function object should return true if the next endpoint should be tried, and false if it should be skipped.

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_context::post()`.

Example

The following connect condition function object can be used to output information about the individual connection attempts:

```
struct my_connect_condition
{
    bool operator()(const asio::error_code& ec,
                     const ::tcp::endpoint& next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next << std::endl;
        return true;
    }
};
```

It would be used with the `asio::connect` function as follows:

```
tcp::resolver r(io_context);
tcp::resolver::query q("host", "service");
tcp::socket s(io_context);

// ...

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;
    }
}

```

5.59 `async_read`

Start an asynchronous operation to read a certain amount of data from a stream.

```

template<
    typename AsyncReadStream,
    typename MutableBufferSequence,
    typename ReadHandler>
DEDUCED async_read(
    AsyncReadStream & s,
    const MutableBufferSequence & buffers,
    ReadHandler && handler,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >::value >::type * = ↵
        0);

template<
    typename AsyncReadStream,
    typename MutableBufferSequence,
    typename CompletionCondition,
    typename ReadHandler>
DEDUCED async_read(
    AsyncReadStream & s,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    ReadHandler && handler,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >::value >::type * = ↵
        0);

template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename ReadHandler>
DEDUCED async_read(
    AsyncReadStream & s,
    DynamicBuffer && buffers,
    ReadHandler && handler,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename CompletionCondition,
    typename ReadHandler>
DEDUCED async_read(

```

```

AsyncReadStream & s,
DynamicBuffer && buffers,
CompletionCondition completion_condition,
ReadHandler && handler,
typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
DEDUCED async_read(
    AsyncReadStream & s,
    basic_streambuf< Allocator > & b,
    ReadHandler && handler);

template<
    typename AsyncReadStream,
    typename Allocator,
    typename CompletionCondition,
    typename ReadHandler>
DEDUCED async_read(
    AsyncReadStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    ReadHandler && handler);

```

Requirements

Header: asio/read.hpp

Convenience header: asio.hpp

5.59.1 `async_read` (1 of 6 overloads)

Start an asynchronous operation to read a certain amount of data from a stream.

```

template<
    typename AsyncReadStream,
    typename MutableBufferSequence,
    typename ReadHandler>
DEDUCED async_read(
    AsyncReadStream & s,
    const MutableBufferSequence & buffers,
    ReadHandler && handler,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >::value >::type * = ↵
        0);

```

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_context::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);
```

5.59.2 `async_read` (2 of 6 overloads)

Start an asynchronous operation to read a certain amount of data from a stream.

```
template<
    typename AsyncReadStream,
    typename MutableBufferSequence,
    typename CompletionCondition,
    typename ReadHandler>
DEDUCED async_read(
    AsyncReadStream & s,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    ReadHandler && handler,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >::value >::type * = ←
        0);
```

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_context::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`.

5.59.3 `async_read` (3 of 6 overloads)

Start an asynchronous operation to read a certain amount of data from a stream.

```
template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename ReadHandler>
DEDUCED async_read(
    AsyncReadStream & s,
    DynamicBuffer && buffers,
    ReadHandler && handler,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 specified dynamic buffer sequence 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.

buffers The dynamic buffer sequence 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 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_context::post()`.

Remarks

This overload is equivalent to calling:

```
asio::async_read(
    s, buffers,
    asio::transfer_all(),
    handler);
```

5.59.4 `async_read` (4 of 6 overloads)

Start an asynchronous operation to read a certain amount of data from a stream.

```
template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename CompletionCondition,
    typename ReadHandler>
DEDUCED async_read(
    AsyncReadStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    ReadHandler && handler,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 specified dynamic buffer sequence 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.

buffers The dynamic buffer sequence 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.

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_context::post()`.

5.59.5 `async_read` (5 of 6 overloads)

Start an asynchronous operation to read a certain amount of data from a stream.

```
template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
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_context::post()`.

Remarks

This overload is equivalent to calling:

```
asio::async_read(
    s, b,
    asio::transfer_all(),
    handler);
```

5.59.6 `async_read` (6 of 6 overloads)

Start an asynchronous operation to read a certain amount of data from a stream.

```
template<
    typename AsyncReadStream,
    typename Allocator,
    typename CompletionCondition,
    typename ReadHandler>
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_context::post()`.

5.60 `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>
DEDUCED async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    ReadHandler && handler);

template<
    typename AsyncRandomAccessReadDevice,
    typename MutableBufferSequence,
    typename CompletionCondition,
    typename ReadHandler>
DEDUCED async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    ReadHandler && handler);

template<
    typename AsyncRandomAccessReadDevice,
    typename Allocator,
    typename ReadHandler>
DEDUCED async_read_at(
    AsyncRandomAccessReadDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    ReadHandler && handler);

template<
    typename AsyncRandomAccessReadDevice,
    typename Allocator,
    typename CompletionCondition,
    typename ReadHandler>
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`

5.60.1 `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>
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_context::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);
```

5.60.2 `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>
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_context::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`.

5.60.3 `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>
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_context::post()`.

Remarks

This overload is equivalent to calling:

```

asio::async_read_at(
    d, 42, b,
    asio::transfer_all(),
    handler);

```

5.60.4 `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>
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_context::post()`.

5.61 `async_read_until`

Start an asynchronous operation to read data into a dynamic buffer sequence, or into a `streambuf`, until it contains a delimiter, matches a regular expression, or a function object indicates a match.

```

template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,
    DynamicBuffer && buffers,
    char delim,
    ReadHandler && handler);

template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,
    DynamicBuffer && buffers,
    string_view delim,
    ReadHandler && handler);

template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,

```

```

DynamicBuffer && buffers,
const boost::regex & expr,
ReadHandler && handler);

template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename MatchCondition,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,
    DynamicBuffer && buffers,
    MatchCondition match_condition,
    ReadHandler && handler,
    typename enable_if< is_match_condition< MatchCondition >::value >::type * = 0);

template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    char delim,
    ReadHandler && handler);

template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    string_view delim,
    ReadHandler && handler);

template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
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>
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

5.61.1 `async_read_until` (1 of 8 overloads)

Start an asynchronous operation to read data into a dynamic buffer sequence until it contains a specified delimiter.

```
template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,
    DynamicBuffer && buffers,
    char delim,
    ReadHandler && handler);
```

This function is used to asynchronously read data into the specified dynamic buffer sequence until the dynamic buffer sequence'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 dynamic buffer sequence 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 dynamic buffer sequence'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.

buffers The dynamic buffer sequence 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.

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 dynamic buffer sequence'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_context::post()`.

Remarks

After a successful `async_read_until` operation, the dynamic buffer sequence may contain additional data beyond the delimiter. An application will typically leave that data in the dynamic buffer sequence for a subsequent `async_read_until` operation to examine.

Example

To asynchronously read data into a `std::string` until a newline is encountered:

```
std::string data;
...
void handler(const asio::error_code& e, std::size_t size)
{
    if (!e)
    {
        std::string line = data.substr(0, n);
        data.erase(0, n);
        ...
    }
}
...
asio::async_read_until(s, data, '\n', handler);
```

After the `async_read_until` operation completes successfully, the buffer `data` contains the delimiter:

```
{ 'a', 'b', ..., 'c', '\n', 'd', 'e', ... }
```

The call to `substr` then extracts the data up to and including the delimiter, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\n' }
```

After the call to `erase`, the remaining data is left in the buffer `data` as follows:

```
{ 'd', 'e', ... }
```

This data may be the start of a new line, to be extracted by a subsequent `async_read_until` operation.

5.61.2 `async_read_until` (2 of 8 overloads)

Start an asynchronous operation to read data into a dynamic buffer sequence until it contains a specified delimiter.

```
template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,
    DynamicBuffer && buffers,
    string_view delim,
    ReadHandler && handler);
```

This function is used to asynchronously read data into the specified dynamic buffer sequence until the dynamic buffer sequence'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 dynamic buffer sequence 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 dynamic buffer sequence'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.

buffers The dynamic buffer sequence 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.

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 dynamic buffer sequence'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_context::post()`.

Remarks

After a successful `async_read_until` operation, the dynamic buffer sequence may contain additional data beyond the delimiter. An application will typically leave that data in the dynamic buffer sequence for a subsequent `async_read_until` operation to examine.

Example

To asynchronously read data into a `std::string` until a CR-LF sequence is encountered:

```
std::string data;  
...  
void handler(const asio::error_code& e, std::size_t size)  
{  
    if (!e)  
    {  
        std::string line = data.substr(0, n);  
        data.erase(0, n);  
        ...  
    }  
    ...  
}  
...  
asio::async_read_until(s, data, "\r\n", handler);
```

After the `async_read_until` operation completes successfully, the string `data` contains the delimiter:

```
{ 'a', 'b', ..., 'c', '\r', '\n', 'd', 'e', ... }
```

The call to `substr` then extracts the data up to and including the delimiter, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r', '\n' }
```

After the call to `erase`, the remaining data is left in the string `data` as follows:

```
{ 'd', 'e', ... }
```

This data may be the start of a new line, to be extracted by a subsequent `async_read_until` operation.

5.61.3 `async_read_until` (3 of 8 overloads)

Start an asynchronous operation to read data into a dynamic buffer sequence until some part of its data matches a regular expression.

```
template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,
    DynamicBuffer && buffers,
    const boost::regex & expr,
    ReadHandler && handler);
```

This function is used to asynchronously read data into the specified dynamic buffer sequence until the dynamic buffer sequence'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 dynamic buffer sequence'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 dynamic buffer sequence'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.

buffers The dynamic buffer sequence 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.

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 dynamic buffer
    // sequence'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_context::post()`.

Remarks

After a successful `async_read_until` operation, the dynamic buffer sequence may contain additional data beyond that which matched the regular expression. An application will typically leave that data in the dynamic buffer sequence for a subsequent `async_read_until` operation to examine.

Example

To asynchronously read data into a `std::string` until a CR-LF sequence is encountered:

```
std::string data;
...
void handler(const asio::error_code& e, std::size_t size)
{
    if (!e)
    {
        std::string line = data.substr(0, n);
        data.erase(0, n);
        ...
    }
}
...
asio::async_read_until(s, data,
    boost::regex("\r\n"), handler);
```

After the `async_read_until` operation completes successfully, the string `data` contains the data which matched the regular expression:

```
{ 'a', 'b', ..., 'c', '\r', '\n', 'd', 'e', ... }
```

The call to `substr` then extracts the data up to and including the match, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r', '\n' }
```

After the call to `erase`, the remaining data is left in the string `data` as follows:

```
{ 'd', 'e', ... }
```

This data may be the start of a new line, to be extracted by a subsequent `async_read_until` operation.

5.61.4 `async_read_until` (4 of 8 overloads)

Start an asynchronous operation to read data into a dynamic buffer sequence until a function object indicates a match.

```
template<
    typename AsyncReadStream,
    typename DynamicBuffer,
    typename MatchCondition,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,
    DynamicBuffer && buffers,
    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 dynamic buffer sequence until a user-defined match condition function object, when applied to the data contained in the dynamic buffer sequence, 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.

buffers The dynamic buffer sequence 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.

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<typename DynamicBuffer::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 dynamic buffer sequence'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_context::post()`.

Remarks

After a successful `async_read_until` operation, the dynamic buffer sequence may contain additional data beyond that which matched the function object. An application will typically leave that data in the dynamic buffer sequence 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 `std::string` until whitespace is encountered:

```
typedef asio::buffers_iterator<
    asio::const_buffers_1> 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);
...
std::string data;
asio::async_read_until(s, data, match whitespace, handler);

```

To asynchronously read data into a `std::string` 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_;
};

namespace asio {
    template <> struct is_match_condition<match_char>
        : public boost::true_type {};
} // namespace asio
...
void handler(const asio::error_code& e, std::size_t size);
...
std::string data;
asio::async_read_until(s, data, match_char('a'), handler);

```

5.61.5 `async_read_until` (5 of 8 overloads)

Start an asynchronous operation to read data into a `streambuf` until it contains a specified delimiter.

```

template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
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_context::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 newline (which is discarded), so that the string `line` contains:

```
{ 'a', 'b', ..., 'c' }
```

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.

5.61.6 `async_read_until` (6 of 8 overloads)

Start an asynchronous operation to read data into a `streambuf` until it contains a specified delimiter.

```
template<
    typename AsyncReadStream,
    typename Allocator,
    typename ReadHandler>
DEDUCED async_read_until(
    AsyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    string_view 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_context::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 newline (which is discarded), so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r' }
```

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.

5.61.7 `async_read_until` (7 of 8 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>
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_context::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 newline (which is discarded), so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r' }
```

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.

5.61.8 `async_read_until` (8 of 8 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>
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_context::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 (*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 {};
} // namespaceasio
...
void handler(constasio::error_code& e, std::size_t size);
...
asio::streambuf b;
asio::async_read_until(s, b, match_char('a')), handler);

```

5.62 `async_result`

An interface for customising the behaviour of an initiating function.

```

template<
    typename CompletionToken,
    typename Signature>
class async_result

```

Types

Name	Description
<code>completion_handler_type</code>	The concrete completion handler type for the specific signature.
<code>return_type</code>	The return type of the initiating function.

Member Functions

Name	Description
<code>async_result</code>	Construct an async result from a given handler.
<code>get</code>	Obtain the value to be returned from the initiating function.

The `async_result` traits class is used for determining:

- the concrete completion handler type to be called at the end of the asynchronous operation;
- the initiating function return type; and
- how the return value of the initiating function is obtained.

The trait allows the handler and return types to be determined at the point where the specific completion handler signature is known.

This template may be specialised for user-defined completion token types. The primary template assumes that the `CompletionToken` is the completion handler.

Requirements

Header: `asio/async_result.hpp`

Convenience header: `asio.hpp`

5.62.1 `async_result::async_result`

Construct an `async result` from a given handler.

```
async_result(  
    completion_handler_type & h);
```

When using a specialised `async_result`, the constructor has an opportunity to initialise some state associated with the completion handler, which is then returned from the initiating function.

5.62.2 `async_result::completion_handler_type`

The concrete completion handler type for the specific signature.

```
typedef CompletionToken completion_handler_type;
```

Requirements

Header: `asio/async_result.hpp`

Convenience header: `asio.hpp`

5.62.3 `async_result::get`

Obtain the value to be returned from the initiating function.

```
return_type get();
```

5.62.4 `async_result::return_type`

The return type of the initiating function.

```
typedef void return_type;
```

Requirements

Header: asio/async_result.hpp

Convenience header: asio.hpp

5.63 `async_result< Handler >`

(Deprecated: Use two-parameter version of `async_result`.) An interface for customising the behaviour of an initiating function.

```
template<
    typename Handler>
class async_result< Handler >
```

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

5.63.1 `async_result< Handler >::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.

5.63.2 `async_result< Handler >::get`

Obtain the value to be returned from the initiating function.

```
type get();
```

5.63.3 `async_result< Handler >::type`

The return type of the initiating function.

```
typedef void type;
```

Requirements

Header: asio/async_result.hpp

Convenience header: asio.hpp

5.64 `async_result< std::packaged_task< Result(Args...) >, Signature >`

Partial specialisation of `async_result` for `std::packaged_task`.

```
template<
    typename Result,
    typename... Args,
    typename Signature>
class async_result< std::packaged_task< Result(Args...) >, Signature >
```

Types

Name	Description
<code>completion_handler_type</code>	The packaged task is the concrete completion handler type.
<code>return_type</code>	The return type of the initiating function is the future obtained from the packaged task.

Member Functions

Name	Description
<code>async_result</code>	The constructor extracts the future from the packaged task.
<code>get</code>	Returns the packaged task's future.

Requirements

Header: asio/packaged_task.hpp

Convenience header: asio.hpp

5.64.1 `async_result< std::packaged_task< Result(Args...) >, Signature >::async_result`

The constructor extracts the future from the packaged task.

```
async_result(
    completion_handler_type & h);
```

5.64.2 `async_result< std::packaged_task< Result(Args...)>, Signature >::completion_handler_type`

The packaged task is the concrete completion handler type.

```
typedef std::packaged_task< Result(Args...)> completion_handler_type;
```

Requirements

Header: asio/packaged_task.hpp

Convenience header: asio.hpp

5.64.3 `async_result< std::packaged_task< Result(Args...)>, Signature >::get`

Returns the packaged task's future.

```
return_type get();
```

5.64.4 `async_result< std::packaged_task< Result(Args...)>, Signature >::return_type`

The return type of the initiating function is the future obtained from the packaged task.

```
typedef std::future< Result > return_type;
```

Requirements

Header: asio/packaged_task.hpp

Convenience header: asio.hpp

5.65 `async_write`

Start an asynchronous operation to write a certain amount of data to a stream.

```
template<
    typename AsyncWriteStream,
    typename ConstBufferSequence,
    typename WriteHandler>
DEDUCED async_write(
    AsyncWriteStream & s,
    const ConstBufferSequence & buffers,
    WriteHandler && handler,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);

template<
    typename AsyncWriteStream,
    typename ConstBufferSequence,
    typename CompletionCondition,
    typename WriteHandler>
DEDUCED async_write(
    AsyncWriteStream & s,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    WriteHandler && handler,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);
```

```

template<
    typename AsyncWriteStream,
    typename DynamicBuffer,
    typename WriteHandler>
DEDUCED async_write(
    AsyncWriteStream & s,
    DynamicBuffer && buffers,
    WriteHandler && handler,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

template<
    typename AsyncWriteStream,
    typename DynamicBuffer,
    typename CompletionCondition,
    typename WriteHandler>
DEDUCED async_write(
    AsyncWriteStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    WriteHandler && handler,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

template<
    typename AsyncWriteStream,
    typename Allocator,
    typename WriteHandler>
DEDUCED async_write(
    AsyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    WriteHandler && handler);

template<
    typename AsyncWriteStream,
    typename Allocator,
    typename CompletionCondition,
    typename WriteHandler>
DEDUCED async_write(
    AsyncWriteStream & s,
    basic_streambuf< Allocator > & b,
    CompletionCondition completion_condition,
    WriteHandler && handler);

```

Requirements

Header: asio/write.hpp

Convenience header: asio.hpp

5.65.1 `async_write` (1 of 6 overloads)

Start an asynchronous operation to write all of the supplied data to a stream.

```
template<
    typename AsyncWriteStream,
```

```

typename ConstBufferSequence,
typename WriteHandler>
DEDUCED async_write(
    AsyncWriteStream & s,
    const ConstBufferSequence & buffers,
    WriteHandler && handler,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);

```

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_context::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`.

5.65.2 `async_write` (2 of 6 overloads)

Start an asynchronous operation to write a certain amount of data to a stream.

```

template<
    typename AsyncWriteStream,
    typename ConstBufferSequence,
    typename CompletionCondition,
    typename WriteHandler>
DEDUCED async_write(
    AsyncWriteStream & s,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    WriteHandler && handler,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);

```

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_context::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`.

5.65.3 `async_write` (3 of 6 overloads)

Start an asynchronous operation to write all of the supplied data to a stream.

```
template<
    typename AsyncWriteStream,
    typename DynamicBuffer,
    typename WriteHandler>
DEDUCED async_write(
    AsyncWriteStream & s,
    DynamicBuffer && buffers,
    WriteHandler && handler,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 dynamic buffer sequence 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.

buffers The dynamic buffer sequence from which data will 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. Successfully written data is automatically consumed from the buffers.

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_context::post()`.

5.65.4 `async_write` (4 of 6 overloads)

Start an asynchronous operation to write a certain amount of data to a stream.

```
template<
    typename AsyncWriteStream,
    typename DynamicBuffer,
    typename CompletionCondition,
    typename WriteHandler>
DEDUCED async_write(
    AsyncWriteStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    WriteHandler && handler,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 dynamic buffer sequence 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.

buffers The dynamic buffer sequence from which data will 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. Successfully written data is automatically consumed from the `buffers`.

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_context::post()`.

5.65.5 `async_write` (5 of 6 overloads)

Start an asynchronous operation to write all of the supplied data to a stream.

```
template<
    typename AsyncWriteStream,
    typename Allocator,
    typename WriteHandler>
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_context::post()`.

5.65.6 `async_write` (6 of 6 overloads)

Start an asynchronous operation to write a certain amount of data to a stream.

```
template<
    typename AsyncWriteStream,
    typename Allocator,
    typename CompletionCondition,
    typename WriteHandler>
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_context::post()`.

5.66 `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>
DEDUCED async_write_at(
    AsyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers,
```

```

        WriteHandler && handler);

template<
    typename AsyncRandomAccessWriteDevice,
    typename ConstBufferSequence,
    typename CompletionCondition,
    typename WriteHandler>
DEDUCED async_write_at(
    AsyncRandomAccessWriteDevice & d,
    uint64_t offset,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    WriteHandler && handler);

template<
    typename AsyncRandomAccessWriteDevice,
    typename Allocator,
    typename WriteHandler>
DEDUCED async_write_at(
    AsyncRandomAccessWriteDevice & d,
    uint64_t offset,
    basic_streambuf< Allocator > & b,
    WriteHandler && handler);

template<
    typename AsyncRandomAccessWriteDevice,
    typename Allocator,
    typename CompletionCondition,
    typename WriteHandler>
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

5.66.1 `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>
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_context::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`.

5.66.2 `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>  
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_context::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`.

5.66.3 `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>
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_context::post()`.

5.66.4 `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>
```

```

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_context::post()`.

5.67 bad_executor

Exception thrown when trying to access an empty polymorphic executor.

```
class bad_executor :  
    public std::exception
```

Member Functions

Name	Description
bad_executor	Constructor.
what	Obtain message associated with exception.

Requirements

Header: asio/executor.hpp

Convenience header: asio.hpp

5.67.1 bad_executor::bad_executor

Constructor.

```
bad_executor();
```

5.67.2 bad_executor::what

Obtain message associated with exception.

```
virtual const char * what() const;
```

5.68 basic_datagram_socket

Provides datagram-oriented socket functionality.

```
template<  
    typename Protocol>  
class basic_datagram_socket :  
    public basic_socket< Protocol >
```

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.

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.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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.
release	Release ownership of the underlying native socket.
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
~basic_datagram_socket	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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);

void 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.

```
void 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>
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_context::post()`.

Example

```
void connect_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Connect succeeded.
    }
}

...

asio::ip::tcp::socket socket(io_context);
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>
DEDUCED async_receive(
    const MutableBufferSequence & buffers,
```

```

ReadHandler && handler);

template<
    typename MutableBufferSequence,
    typename ReadHandler>
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>
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_context::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>
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_context::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>
DEDUCED async_receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    ReadHandler && handler);

template<
    typename MutableBufferSequence,
    typename ReadHandler>
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>
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_context::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>
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_context::post()`.

5.68.5 basic_datagram_socket::async_send

Start an asynchronous send on a connected socket.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
DEDUCED async_send(
    const ConstBufferSequence & buffers,
    WriteHandler && handler);

template<
    typename ConstBufferSequence,
    typename WriteHandler>
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>
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_context::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>
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_context::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>
DEDUCED async_send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    WriteHandler && handler);

template<
    typename ConstBufferSequence,
    typename WriteHandler>
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>
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_context::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>
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_context::post()`.

5.68.7 basic_datagram_socket::async_wait

Inherited from basic_socket.

Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
template<
    typename WaitHandler>
DEDUCED async_wait(
    wait_type w,
    WaitHandler && handler);
```

This function is used to perform an asynchronous wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

handler The handler to be called when the wait 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_context::post()`.

Example

```
void wait_handler(const asio::error_code& error)  
{  
    if (!error)  
    {  
        // Wait succeeded.  
    }  
}  
  
...  
  
asio::ip::tcp::socket socket(io_context);  
...  
socket.async_wait(asio::ip::tcp::socket::wait_read, wait_handler);
```

5.68.8 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;
```

5.68.8.1 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.

5.68.8.2 `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.

5.68.9 `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;
```

5.68.9.1 `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.

5.68.9.2 `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.

5.68.10 `basic_datagram_socket::basic_datagram_socket`

Construct a `basic_datagram_socket` without opening it.

```
explicit basic_datagram_socket(  
    asio::io_context & io_context);
```

Construct and open a `basic_datagram_socket`.

```
basic_datagram_socket(  
    asio::io_context & io_context,  
    const protocol_type & protocol);
```

Construct a `basic_datagram_socket`, opening it and binding it to the given local endpoint.

```
basic_datagram_socket(  
    asio::io_context & io_context,  
    const endpoint_type & endpoint);
```

Construct a `basic_datagram_socket` on an existing native socket.

```
basic_datagram_socket(  
    asio::io_context & io_context,  
    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>  
basic_datagram_socket(  
    basic_datagram_socket< Protocol1 > && other,  
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.68.10.1 basic_datagram_socket::basic_datagram_socket (1 of 6 overloads)

Construct a `basic_datagram_socket` without opening it.

```
basic_datagram_socket(
    asio::io_context & io_context);
```

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_context The `io_context` object that the datagram socket will use to dispatch handlers for any asynchronous operations performed on the socket.

5.68.10.2 basic_datagram_socket::basic_datagram_socket (2 of 6 overloads)

Construct and open a `basic_datagram_socket`.

```
basic_datagram_socket(
    asio::io_context & io_context,
    const protocol_type & protocol);
```

This constructor creates and opens a datagram socket.

Parameters

io_context The `io_context` 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.

5.68.10.3 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_context & io_context,
    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_context The `io_context` 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.

5.68.10.4 `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_context & io_context,
    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_context The `io_context` 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.

5.68.10.5 `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_context &)` constructor.

5.68.10.6 `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>
basic_datagram_socket(
    basic_datagram_socket< Protocol1 > && 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_context &)` constructor.

5.68.11 `basic_datagram_socket::bind`

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

void bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.68.11.1 `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_context);
socket.open(asio::ip::tcp::v4());
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345));
```

5.68.11.2 basic_datagram_socket::bind (2 of 2 overloads)

Inherited from `basic_socket`.

Bind the socket to the given local endpoint.

```
void 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_context);
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.
}
```

5.68.12 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_context);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.68.13 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_context);
...
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

5.68.14 basic_datagram_socket::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();

void cancel(
    asio::error_code & ec);
```

5.68.14.1 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.

5.68.14.2 basic_datagram_socket::cancel (2 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
void 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.

5.68.15 basic_datagram_socket::close

Close the socket.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.68.15.1 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.

5.68.15.2 basic_datagram_socket::close (2 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
void 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_context);  
...  
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.

5.68.16 `basic_datagram_socket::connect`

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);

void connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.68.16.1 `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_context);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.connect(endpoint);
```

5.68.16.2 basic_datagram_socket::connect (2 of 2 overloads)

Inherited from `basic_socket`.

Connect the socket to the specified endpoint.

```
void 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_context);
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.
}
```

5.68.17 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_context);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.68.18 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_context);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.68.19 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_context);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.68.20 basic_datagram_socket::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.21 basic_datagram_socket::executor_type

Inherited from basic_socket.

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.

Name	Description
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.22 basic_datagram_socket::get_executor

Inherited from basic_socket.

Get the executor associated with the object.

```
executor_type get_executor();
```

5.68.23 basic_datagram_socket::get_io_context

Inherited from basic_socket.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.68.24 basic_datagram_socket::get_io_service

Inherited from basic_socket.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.68.25 `basic_datagram_socket::get_option`

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;

template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.68.25.1 `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_context);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.68.25.2 basic_datagram_socket::get_option (2 of 2 overloads)

Inherited from `basic_socket`.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void 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_context);
...
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();
```

5.68.26 basic_datagram_socket::io_control

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
void 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_context);
...
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>
void 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_context);
...
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_context);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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_context);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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_context);
...
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_context);
...
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 > 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>executor_type</code>	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</code> associated with the object.
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</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_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>release</code>	Release ownership of the underlying native socket.

Name	Description
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

Protected Member Functions

Name	Description
<code>~basic_socket</code>	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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.

(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.68.34 basic_datagram_socket::max_listen_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_listen_connections = implementation_defined;
```

5.68.35 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;
```

5.68.36 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;
```

5.68.37 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

5.68.38 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;
```

5.68.39 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;
```

5.68.40 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 implementation_defined 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);
```

```
void 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, 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.

```
void 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, op);
}

```

5.68.43 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);

void non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.68.43.1 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`.

5.68.43.2 `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`.

5.68.43.3 `basic_datagram_socket::non_blocking` (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
void 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`.

5.68.44 basic_datagram_socket::open

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());  
  
void open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.68.44.1 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_context);
socket.open(asio::ip::tcp::v4());
```

5.68.44.2 basic_datagram_socket::open (2 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
void 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_context);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.68.45 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>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_datagram_socket >::type & ←
operator=(
    basic_datagram_socket< Protocol1 > && other);
```

5.68.45.1 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_context &)` constructor.

5.68.45.2 basic_datagram_socket::operator= (2 of 2 overloads)

Move-assign a `basic_datagram_socket` from a socket of another protocol type.

```
template<
    typename Protocol1>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_datagram_socket >::type & ←
operator=(
    basic_datagram_socket< Protocol1 > && 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_context&)` constructor.

5.68.46 `basic_datagram_socket::out_of_band_inline`

Inherited from socket_base.

Socket option for putting received out-of-band data inline.

```
typedef implementation_defined out_of_band_inline;
```

Implements the SOL_SOCKET/SO_OOBINLINE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option;
socket.get_option(option);
bool value = option.value();
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.47 `basic_datagram_socket::protocol_type`

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_datagram_socket.hpp

Convenience header: asio.hpp

5.68.48 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);
```

5.68.48.1 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.

5.68.48.2 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.

5.68.48.3 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.

5.68.49 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_context);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.68.50 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);
```

5.68.50.1 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.

5.68.50.2 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.

5.68.50.3 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.

5.68.51 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_context);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.68.52 basic_datagram_socket::release

Release ownership of the underlying native socket.

```
native_handle_type release();
```

```
native_handle_type release(
    asio::error_code & ec);
```

5.68.52.1 basic_datagram_socket::release (1 of 2 overloads)

Inherited from basic_socket.

Release ownership of the underlying native socket.

```
native_handle_type release();
```

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. Ownership of the native socket is then transferred to the caller.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

5.68.52.2 basic_datagram_socket::release (2 of 2 overloads)

Inherited from basic_socket.

Release ownership of the underlying native socket.

```
native_handle_type release(
    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. Ownership of the native socket is then transferred to the caller.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

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_context);
...
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_context);
...
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_context);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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_context);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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_context);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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::set_option

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);

template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.68.59.1 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_context);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.68.59.2 basic_datagram_socket::set_option (2 of 2 overloads)

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void 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_context);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.68.60 basic_datagram_socket::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

void shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.68.60.1 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_context);  
...  
socket.shutdown(asio::ip::tcp::socket::shutdown_send);
```

5.68.60.2 basic_datagram_socket::shutdown (2 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
void 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_context);  
...  
asio::error_code ec;  
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);  
if (ec)  
{  
    // An error occurred.  
}
```

5.68.61 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.

5.68.62 basic_datagram_socket::wait

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);  
  
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

5.68.62.1 basic_datagram_socket::wait (1 of 2 overloads)

Inherited from basic_socket.

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);
```

This function is used to perform a blocking wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

Example

Waiting for a socket to become readable.

```
asio::ip::tcp::socket socket(io_context);  
...  
socket.wait(asio::ip::tcp::socket::wait_read);
```

5.68.62.2 basic_datagram_socket::wait (2 of 2 overloads)

Inherited from `basic_socket`.

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

This function is used to perform a blocking wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

ec Set to indicate what error occurred, if any.

Example

Waiting for a socket to become readable.

```
asio::ip::tcp::socket socket(io_context);  
...  
asio::error_code ec;  
socket.wait(asio::ip::tcp::socket::wait_read, ec);
```

5.68.63 basic_datagram_socket::wait_type

Inherited from `socket_base`.

Wait types.

```
enum wait_type
```

Values

wait_read Wait for a socket to become ready to read.

wait_write Wait for a socket to become ready to write.

wait_error Wait for a socket to have error conditions pending.

For use with `basic_socket::wait()` and `basic_socket::async_wait()`.

5.68.64 basic_datagram_socket::~basic_datagram_socket

Destroys the socket.

```
~basic_datagram_socket();
```

This function destroys the socket, cancelling any outstanding asynchronous operations associated with the socket as if by calling `cancel`.

5.69 basic_deadline_timer

Provides waitable timer functionality.

```
template<
    typename Time,
    typename TimeTraits = asio::time_traits<Time>>
class basic_deadline_timer
```

Types

Name	Description
duration_type	The duration type.
executor_type	The type of the executor associated with the object.
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. Move-construct a basic_deadline_timer from another.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context associated with the object.
operator=	Move-assign a basic_deadline_timer from another.

Name	Description
wait	Perform a blocking wait on the timer.
<code>~basic_deadline_timer</code>	Destroys the timer.

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_context);

// 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_context,
    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:

```

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>
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_context::post()`.

5.69.2 basic_deadline_timer::basic_deadline_timer

Constructor.

```
explicit basic_deadline_timer(
    asio::io_context & io_context);
```

Constructor to set a particular expiry time as an absolute time.

```
basic_deadline_timer(
    asio::io_context & io_context,
    const time_type & expiry_time);
```

Constructor to set a particular expiry time relative to now.

```
basic_deadline_timer(
    asio::io_context & io_context,
    const duration_type & expiry_time);
```

Move-construct a `basic_deadline_timer` from another.

```
basic_deadline_timer(
    basic_deadline_timer && other);
```

5.69.2.1 basic_deadline_timer::basic_deadline_timer (1 of 4 overloads)

Constructor.

```
basic_deadline_timer(
    asio::io_context & io_context);
```

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_context The `io_context` 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 4 overloads)

Constructor to set a particular expiry time as an absolute time.

```
basic_deadline_timer(
    asio::io_context & io_context,
    const time_type & expiry_time);
```

This constructor creates a timer and sets the expiry time.

Parameters

io_context The `io_context` 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 4 overloads)

Constructor to set a particular expiry time relative to now.

```
basic_deadline_timer(
    asio::io_context & io_context,
    const duration_type & expiry_time);
```

This constructor creates a timer and sets the expiry time.

Parameters

io_context The `io_context` 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.

5.69.2.4 basic_deadline_timer::basic_deadline_timer (4 of 4 overloads)

Move-construct a `basic_deadline_timer` from another.

```
basic_deadline_timer(
    basic_deadline_timer && other);
```

This constructor moves a timer from one object to another.

Parameters

other The other `basic_deadline_timer` 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_deadline_timer(io_context &)` constructor.

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::executor_type`

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
<code>context</code>	Obtain the underlying execution context.
<code>defer</code>	Request the <code>io_context</code> to invoke the given function object.
<code>dispatch</code>	Request the <code>io_context</code> to invoke the given function object.
<code>on_work_finished</code>	Inform the <code>io_context</code> that some work is no longer outstanding.
<code>on_work_started</code>	Inform the <code>io_context</code> that it has some outstanding work to do.
<code>post</code>	Request the <code>io_context</code> to invoke the given function object.
<code>running_in_this_thread</code>	Determine whether the <code>io_context</code> is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/basic_deadline_timer.hpp

Convenience header: asio.hpp

5.69.7 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);
```

5.69.7.1 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.

5.69.7.2 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.

5.69.7.3 `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.

5.69.8 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);
```

5.69.8.1 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.

5.69.8.2 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.

5.69.8.3 `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.

5.69.9 `basic_deadline_timer::get_executor`

Get the executor associated with the object.

```
executor_type get_executor();
```

5.69.10 `basic_deadline_timer::get_io_context`

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.69.11 basic_deadline_timer::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.69.12 basic_deadline_timer::operator=

Move-assign a `basic_deadline_timer` from another.

```
basic_deadline_timer & operator=(  
    basic_deadline_timer && other);
```

This assignment operator moves a timer from one object to another. Cancels any outstanding asynchronous operations associated with the target object.

Parameters

other The other `basic_deadline_timer` 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_deadline_timer(io_context&)` constructor.

5.69.13 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`

5.69.14 basic_deadline_timer::traits_type

The time traits type.

```
typedef TimeTraits traits_type;
```

Requirements

Header: `asio/basic_deadline_timer.hpp`

Convenience header: `asio.hpp`

5.69.15 basic_deadline_timer::wait

Perform a blocking wait on the timer.

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

5.69.15.1 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.

5.69.15.2 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.

5.69.16 basic_deadline_timer::~basic_deadline_timer

Destroys the timer.

```
~basic_deadline_timer();
```

This function destroys the timer, cancelling any outstanding asynchronous wait operations associated with the timer as if by calling `cancel`.

5.70 basic_io_object

Base class for all I/O objects.

```
template<  
    typename IoObjectService>  
class basic_io_object
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context associated with the object.

Protected Member Functions

Name	Description
basic_io_object	Construct a basic_io_object. Move-construct a basic_io_object. Perform a converting move-construction of 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.

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_context & io_context);
```

Move-construct a **basic_io_object**.

```
basic_io_object(
    basic_io_object && other);
```

Perform a converting move-construction of a **basic_io_object**.

```
template<
    typename IoObjectService1>
basic_io_object(
    IoObjectService1 & other_service,
    typename IoObjectService1::implementation_type & other_implementation);
```

5.70.1.1 **basic_io_object::basic_io_object (1 of 3 overloads)**

Construct a **basic_io_object**.

```
basic_io_object(
    asio::io_context & io_context);
```

Performs:

```
get_service().construct(get_implementation());
```

5.70.1.2 **basic_io_object::basic_io_object (2 of 3 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,

5.70.1.3 **basic_io_object::basic_io_object (3 of 3 overloads)**

Perform a converting move-construction of a **basic_io_object**.

```
template<
    typename IoObjectService1>
basic_io_object(
    IoObjectService1 & other_service,
    typename IoObjectService1::implementation_type & other_implementation);
```

5.70.2 basic_io_object::executor_type

The type of the executor associated with the object.

```
typedef asio::io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/basic_io_object.hpp

Convenience header: asio.hpp

5.70.3 basic_io_object::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.70.4 basic_io_object::get_implementation

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();  
  
const implementation_type & get_implementation() const;
```

5.70.4.1 basic_io_object::get_implementation (1 of 2 overloads)

Get the underlying implementation of the I/O object.

```
implementation_type & get_implementation();
```

5.70.4.2 basic_io_object::get_implementation (2 of 2 overloads)

Get the underlying implementation of the I/O object.

```
const implementation_type & get_implementation() const;
```

5.70.5 basic_io_object::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.70.6 basic_io_object::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.70.7 basic_io_object::get_service

Get the service associated with the I/O object.

```
service_type & get_service();
```

```
const service_type & get_service() const;
```

5.70.7.1 basic_io_object::get_service (1 of 2 overloads)

Get the service associated with the I/O object.

```
service_type & get_service();
```

5.70.7.2 basic_io_object::get_service (2 of 2 overloads)

Get the service associated with the I/O object.

```
const service_type & get_service() const;
```

5.70.8 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

5.70.9 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,

5.70.10 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

5.70.11 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>
class basic_raw_socket :
    public basic_socket< Protocol >
```

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.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
protocol_type	The protocol type.

Name	Description
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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.

Name	Description
close	Close the socket.
connect	Connect the socket to the specified endpoint.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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_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.
release	Release ownership of the underlying native socket.
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.

Name	Description
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
<code>~basic_raw_socket</code>	Destroys the socket.

Data Members

Name	Description
<code>max_connections</code>	(Deprecated: Use <code>max_listen_connections</code> .) The maximum length of the queue of pending incoming connections.
<code>max_listen_connections</code>	The maximum length of the queue of pending incoming connections.
<code>message_do_not_route</code>	Specify that the data should not be subject to routing.
<code>message_end_of_record</code>	Specifies that the data marks the end of a record.
<code>message_out_of_band</code>	Process out-of-band data.
<code>message_peek</code>	Peek at incoming data without removing it from the input queue.

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);

void 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.

```
void 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>
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_context::post()`.

Example

```
void connect_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Connect succeeded.
    }
}

...

asio::ip::tcp::socket socket(io_context);
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>
DEDUCED async_receive(
    const MutableBufferSequence & buffers,
    ReadHandler && handler);

template<
    typename MutableBufferSequence,
    typename ReadHandler>
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>
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_context::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>
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_context::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>
DEDUCED async_receive_from(
    const MutableBufferSequence & buffers,
    endpoint_type & sender_endpoint,
    ReadHandler && handler);

template<
    typename MutableBufferSequence,
    typename ReadHandler>
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>
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_context::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>
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_context::post()`.

5.71.5 basic_raw_socket::async_send

Start an asynchronous send on a connected socket.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
DEDUCED async_send(
    const ConstBufferSequence & buffers,
    WriteHandler && handler);
```

```

template<
    typename ConstBufferSequence,
    typename WriteHandler>
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>
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_context::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>
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_context::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>
DEDUCED async_send_to(
    const ConstBufferSequence & buffers,
    const endpoint_type & destination,
    WriteHandler && handler);

template<
    typename ConstBufferSequence,
    typename WriteHandler>
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>
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_context::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>
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_context::post()`.

5.71.7 basic_raw_socket::async_wait

Inherited from basic_socket.

Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
template<  
    typename WaitHandler>  
DEDUCED async_wait(  
    wait_type w,  
    WaitHandler && handler);
```

This function is used to perform an asynchronous wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

handler The handler to be called when the wait 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_context::post()`.

Example

```
void wait_handler(const asio::error_code& error)  
{  
    if (!error)  
    {  
        // Wait succeeded.  
    }  
}
```

```
...
asio::ip::tcp::socket socket(io_context);
...
socket.async_wait(asio::ip::tcp::socket::wait_read, wait_handler);
```

5.71.8 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;
```

5.71.8.1 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.

5.71.8.2 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.

5.71.9 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;
```

5.71.9.1 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.

5.71.9.2 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.

5.71.10 basic_raw_socket::basic_raw_socket

Construct a `basic_raw_socket` without opening it.

```
explicit basic_raw_socket(
    asio::io_context & io_context);
```

Construct and open a `basic_raw_socket`.

```
basic_raw_socket(
    asio::io_context & io_context,
    const protocol_type & protocol);
```

Construct a `basic_raw_socket`, opening it and binding it to the given local endpoint.

```
basic_raw_socket(
    asio::io_context & io_context,
    const endpoint_type & endpoint);
```

Construct a `basic_raw_socket` on an existing native socket.

```
basic_raw_socket(
    asio::io_context & io_context,
    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>
basic_raw_socket(
    basic_raw_socket< Protocol1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.71.10.1 basic_raw_socket::basic_raw_socket (1 of 6 overloads)

Construct a `basic_raw_socket` without opening it.

```
basic_raw_socket(
    asio::io_context & io_context);
```

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_context The `io_context` object that the raw socket will use to dispatch handlers for any asynchronous operations performed on the socket.

5.71.10.2 basic_raw_socket::basic_raw_socket (2 of 6 overloads)

Construct and open a `basic_raw_socket`.

```
basic_raw_socket(
    asio::io_context & io_context,
    const protocol_type & protocol);
```

This constructor creates and opens a raw socket.

Parameters

io_context The `io_context` 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.

5.71.10.3 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_context & io_context,
    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_context The `io_context` 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.

5.71.10.4 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_context & io_context,
    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_context The `io_context` 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.

5.71.10.5 `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_context &)` constructor.

5.71.10.6 `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>
basic_raw_socket(
    basic_raw_socket< Protocol1 > && 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_context &)` constructor.

5.71.11 basic_raw_socket::bind

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

void bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.71.11.1 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_context);
socket.open(asio::ip::tcp::v4());
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345));
```

5.71.11.2 basic_raw_socket::bind (2 of 2 overloads)

Inherited from basic_socket.

Bind the socket to the given local endpoint.

```
void 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_context);
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.
}
```

5.71.12 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_context);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.71.13 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_context);
...
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

5.71.14 basic_raw_socket::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();

void cancel(
    asio::error_code & ec);
```

5.71.14.1 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.

5.71.14.2 basic_raw_socket::cancel (2 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
void 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.

5.71.15 basic_raw_socket::close

Close the socket.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.71.15.1 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.

5.71.15.2 `basic_raw_socket::close` (2 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
void 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_context);
...
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.

5.71.16 `basic_raw_socket::connect`

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);

void connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.71.16.1 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_context);  
asio::ip::tcp::endpoint endpoint(  
    asio::ip::address::from_string("1.2.3.4"), 12345);  
socket.connect(endpoint);
```

5.71.16.2 basic_raw_socket::connect (2 of 2 overloads)

Inherited from basic_socket.

Connect the socket to the specified endpoint.

```
void 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_context);
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.
}
```

5.71.17 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_context);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.71.18 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_context);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.71.19 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_context);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.71.20 basic_raw_socket::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.21 basic_raw_socket::executor_type

Inherited from basic_socket.

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.22 basic_raw_socket::get_executor

Inherited from basic_socket.

Get the executor associated with the object.

```
executor_type get_executor();
```

5.71.23 basic_raw_socket::get_io_context

Inherited from basic_socket.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.71.24 basic_raw_socket::get_io_service

Inherited from basic_socket.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.71.25 basic_raw_socket::get_option

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;

template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.71.25.1 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_context);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.71.25.2 basic_raw_socket::get_option (2 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void 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_context);
...
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();
```

5.71.26 basic_raw_socket::io_control

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
void 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_context);
...
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>
void 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_context);
...
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_context);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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_context);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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_context);  
...  
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_context);
...
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 > 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>executor_type</code>	The type of the executor associated with the 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>out_of_band_inline</code>	Socket option for putting received out-of-band data inline.
<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.
<code>send_low_watermark</code>	Socket option for the send low watermark.
<code>shutdown_type</code>	Different ways a socket may be shutdown.

Name	Description
wait_type	Wait types.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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.
release	Release ownership of the underlying native socket.
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

Protected Member Functions

Name	Description
<code>~basic_socket</code>	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
<code>max_connections</code>	(Deprecated: Use <code>max_listen_connections</code> .) The maximum length of the queue of pending incoming connections.
<code>max_listen_connections</code>	The maximum length of the queue of pending incoming connections.
<code>message_do_not_route</code>	Specify that the data should not be subject to routing.
<code>message_end_of_record</code>	Specifies that the data marks the end of a record.

Name	Description
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

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.

(Deprecated: Use `max_listen_connections`.) The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.71.34 basic_raw_socket::max_listen_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_listen_connections = implementation_defined;
```

5.71.35 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;
```

5.71.36 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;
```

5.71.37 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

5.71.38 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;
```

5.71.39 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;
```

5.71.40 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 implementation_defined 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);

void 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, 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.

```
void 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, op);
}

```

5.71.43 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);

void non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.71.43.1 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`.

5.71.43.2 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`.

5.71.43.3 `basic_raw_socket::non_blocking` (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
void 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`.

5.71.44 `basic_raw_socket::open`

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());  
  
void open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.71.44.1 `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_context);
socket.open(asio::ip::tcp::v4());
```

5.71.44.2 basic_raw_socket::open (2 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
void 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_context);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.71.45 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 Protocol1>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_raw_socket >::type & operator=(
    basic_raw_socket< Protocol1 > && other);
```

5.71.45.1 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_context&)` constructor.

5.71.45.2 basic_raw_socket::operator=(2 of 2 overloads)

Move-assign a `basic_raw_socket` from a socket of another protocol type.

```
template<
    typename Protocol1>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_raw_socket >::type & operator=(
    basic_raw_socket< Protocol1 > && 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_context&)` constructor.

5.71.46 basic_raw_socket::out_of_band_inline

Inherited from socket_base.

Socket option for putting received out-of-band data inline.

```
typedef implementation_defined out_of_band_inline;
```

Implements the SOL_SOCKET/SO_OOBINLINE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option;
socket.get_option(option);
bool value = option.value();
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.47 basic_raw_socket::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_raw_socket.hpp

Convenience header: asio.hpp

5.71.48 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);
```

5.71.48.1 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`.

5.71.48.2 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.

5.71.48.3 `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.

5.71.49 `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_context);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.71.50 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);
```

5.71.50.1 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`.

5.71.50.2 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.

5.71.50.3 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.

5.71.51 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_context);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.71.52 basic_raw_socket::release

Release ownership of the underlying native socket.

```
native_handle_type release();  
  
native_handle_type release(  
    asio::error_code & ec);
```

5.71.52.1 basic_raw_socket::release (1 of 2 overloads)

Inherited from basic_socket.

Release ownership of the underlying native socket.

```
native_handle_type release();
```

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. Ownership of the native socket is then transferred to the caller.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

5.71.52.2 basic_raw_socket::release (2 of 2 overloads)

Inherited from basic_socket.

Release ownership of the underlying native socket.

```
native_handle_type release(  
    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. Ownership of the native socket is then transferred to the caller.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

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_context);  
...  
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_context);
...
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_context);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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_context);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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_context);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```

asio::ip::tcp::socket socket(io_context);
...
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::set_option

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);

template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.71.59.1 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_context);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.71.59.2 basic_raw_socket::set_option (2 of 2 overloads)

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void 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_context);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.71.60 basic_raw_socket::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

void shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.71.60.1 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_context);
...
socket.shutdown(asio::ip::tcp::socket::shutdown_send);
```

5.71.60.2 basic_raw_socket::shutdown (2 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
void 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_context);
...
asio::error_code ec;
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);
if (ec)
{
    // An error occurred.
}
```

5.71.61 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.

5.71.62 basic_raw_socket::wait

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(
    wait_type w);

void wait(
    wait_type w,
    asio::error_code & ec);
```

5.71.62.1 basic_raw_socket::wait (1 of 2 overloads)

Inherited from basic_socket.

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(
    wait_type w);
```

This function is used to perform a blocking wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

Example

Waiting for a socket to become readable.

```
asio::ip::tcp::socket socket(io_context);
...
socket.wait(asio::ip::tcp::socket::wait_read);
```

5.71.62.2 basic_raw_socket::wait (2 of 2 overloads)

Inherited from basic_socket.

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(
    wait_type w,
    asio::error_code & ec);
```

This function is used to perform a blocking wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

ec Set to indicate what error occurred, if any.

Example

Waiting for a socket to become readable.

```
asio::ip::tcp::socket socket(io_context);
...
asio::error_code ec;
socket.wait(asio::ip::tcp::socket::wait_read, ec);
```

5.71.63 basic_raw_socket::wait_type

Inherited from socket_base.

Wait types.

```
enum wait_type
```

Values

wait_read Wait for a socket to become ready to read.

wait_write Wait for a socket to become ready to write.

wait_error Wait for a socket to have error conditions pending.

For use with `basic_socket::wait()` and `basic_socket::async_wait()`.

5.71.64 basic_raw_socket::~basic_raw_socket

Destroys the socket.

```
~basic_raw_socket();
```

This function destroys the socket, cancelling any outstanding asynchronous operations associated with the socket as if by calling `cancel`.

5.72 basic_seq_packet_socket

Provides sequenced packet socket functionality.

```
template<
    typename Protocol>
class basic_seq_packet_socket :
    public basic_socket< Protocol >
```

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.
executor_type	The type of the executor associated with the 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 <code>basic_socket</code> 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.

Name	Description
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor	Get the executor associated with the object.

Name	Description
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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_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.
release	Release ownership of the underlying native 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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
~basic_seq_packet_socket	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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);

void 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.

```
void 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>
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_context::post()`.

Example

```
void connect_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Connect succeeded.
    }
}

...

asio::ip::tcp::socket socket(io_context);
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>
DEDUCED async_receive(
    const MutableBufferSequence & buffers,
    socket_base::message_flags & out_flags,
    ReadHandler && handler);

template<
    typename MutableBufferSequence,
    typename ReadHandler>
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>
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_flags::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_context::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>
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_flags::record_available` 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_context::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>
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_context::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::async_wait

Inherited from basic_socket.

Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
template<
    typename WaitHandler>
DEDUCED async_wait(
    wait_type w,
    WaitHandler && handler);
```

This function is used to perform an asynchronous wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

handler The handler to be called when the wait 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_context::post()`.

Example

```
void wait_handler(const asio::error_code& error)  
{  
    if (!error)  
    {  
        // Wait succeeded.  
    }  
}  
  
...  
  
asio::ip::tcp::socket socket(io_context);  
...  
socket.async_wait(asio::ip::tcp::socket::wait_read, wait_handler);
```

5.72.6 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;
```

5.72.6.1 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.

5.72.6.2 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.

5.72.7 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;
```

5.72.7.1 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.

5.72.7.2 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.

5.72.8 basic_seq_packet_socket::basic_seq_packet_socket

Construct a `basic_seq_packet_socket` without opening it.

```
explicit basic_seq_packet_socket(  
    asio::io_context & io_context);
```

Construct and open a `basic_seq_packet_socket`.

```
basic_seq_packet_socket(  
    asio::io_context & io_context,  
    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_context & io_context,  
    const endpoint_type & endpoint);
```

Construct a `basic_seq_packet_socket` on an existing native socket.

```
basic_seq_packet_socket(  
    asio::io_context & io_context,  
    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>  
basic_seq_packet_socket(  
    basic_seq_packet_socket< Protocol1 > && other,  
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.72.8.1 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_context & io_context);
```

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_context The `io_context` object that the sequenced packet socket will use to dispatch handlers for any asynchronous operations performed on the socket.

5.72.8.2 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_context & io_context,
   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_context The `io_context` 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.

5.72.8.3 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_context & io_context,
   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_context The `io_context` 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.

5.72.8.4 `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_context & io_context,
    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_context The `io_context` 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.

5.72.8.5 `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_context&)` constructor.

5.72.8.6 `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>
basic_seq_packet_socket(
    basic_seq_packet_socket< Protocol1 > && 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_context&)` constructor.

5.72.9 `basic_seq_packet_socket::bind`

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

void bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.72.9.1 `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_context);
socket.open(asio::ip::tcp::v4());
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345));
```

5.72.9.2 basic_seq_packet_socket::bind (2 of 2 overloads)

Inherited from `basic_socket`.

Bind the socket to the given local endpoint.

```
void 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_context);
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.
}
```

5.72.10 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_context);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.72.11 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_context);
...
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

5.72.12 basic_seq_packet_socket::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();
void cancel(
    asio::error_code & ec);
```

5.72.12.1 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.

5.72.12.2 `basic_seq_packet_socket::cancel (2 of 2 overloads)`

Inherited from `basic_socket`.

Cancel all asynchronous operations associated with the socket.

```
void 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.

5.72.13 basic_seq_packet_socket::close

Close the socket.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.72.13.1 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.

5.72.13.2 basic_seq_packet_socket::close (2 of 2 overloads)

Inherited from basic_socket.

Close the socket.

```
void 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_context);  
...  
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.

5.72.14 basic_seq_packet_socket::connect

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);

void connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.72.14.1 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_context);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.connect(endpoint);
```

5.72.14.2 basic_seq_packet_socket::connect (2 of 2 overloads)

Inherited from `basic_socket`.

Connect the socket to the specified endpoint.

```
void 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_context);
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.
}
```

5.72.15 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_context);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.72.16 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_context);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.72.17 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_context);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.72.18 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

5.72.19 basic_seq_packet_socket::executor_type

Inherited from basic_socket.

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.

Name	Description
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.20 basic_seq_packet_socket::get_executor

Inherited from basic_socket.

Get the executor associated with the object.

```
executor_type get_executor();
```

5.72.21 basic_seq_packet_socket::get_io_context

Inherited from basic_socket.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.72.22 basic_seq_packet_socket::get_io_service

Inherited from basic_socket.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.72.23 `basic_seq_packet_socket::get_option`

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;

template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.72.23.1 `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_context);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.72.23.2 basic_seq_packet_socket::get_option (2 of 2 overloads)

Inherited from `basic_socket`.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void 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_context);
...
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();
```

5.72.24 basic_seq_packet_socket::io_control

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
void 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_context);
...
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>
void 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_context);
...
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_context);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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_context);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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_context);
...
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_context);
...
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 > 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.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</code> associated with the object.
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</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_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>release</code>	Release ownership of the underlying native socket.

Name	Description
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

Protected Member Functions

Name	Description
<code>~basic_socket</code>	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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.

(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.72.32 basic_seq_packet_socket::max_listen_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_listen_connections = implementation_defined;
```

5.72.33 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;
```

5.72.34 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;
```

5.72.35 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

5.72.36 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;
```

5.72.37 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;
```

5.72.38 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 implementation_defined 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);
```

```
void 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, 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.

```
void 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, op);
}

```

5.72.41 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);

void non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.72.41.1 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`.

5.72.41.2 `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`.

5.72.41.3 `basic_seq_packet_socket::non_blocking` (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
void 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`.

5.72.42 basic_seq_packet_socket::open

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());  
  
void open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.72.42.1 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_context);
socket.open(asio::ip::tcp::v4());
```

5.72.42.2 basic_seq_packet_socket::open (2 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
void 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_context);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.72.43 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 Protocol1>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_seq_packet_socket >::type & ←
operator=(
    basic_seq_packet_socket< Protocol1 > && other);
```

5.72.43.1 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_context&)` constructor.

5.72.43.2 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 Protocol1>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_seq_packet_socket >::type & ←
operator=(
    basic_seq_packet_socket< Protocol1 > && 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_context&)` constructor.

5.72.44 `basic_seq_packet_socket::out_of_band_inline`

Inherited from socket_base.

Socket option for putting received out-of-band data inline.

```
typedef implementation_defined out_of_band_inline;
```

Implements the SOL_SOCKET/SO_OOBINLINE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option;
socket.get_option(option);
bool value = option.value();
```

Requirements

Header: asio/basic_seq_packet_socket.hpp

Convenience header: asio.hpp

5.72.45 `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

5.72.46 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);
```

5.72.46.1 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` 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.

5.72.46.2 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` 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.

5.72.46.3 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.

5.72.47 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_context);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.72.48 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_context);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.72.49 basic_seq_packet_socket::release

Release ownership of the underlying native socket.

```
native_handle_type release();

native_handle_type release(
    asio::error_code & ec);
```

5.72.49.1 basic_seq_packet_socket::release (1 of 2 overloads)

Inherited from basic_socket.

Release ownership of the underlying native socket.

```
native_handle_type release();
```

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. Ownership of the native socket is then transferred to the caller.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

5.72.49.2 `basic_seq_packet_socket::release (2 of 2 overloads)`

Inherited from basic_socket.

Release ownership of the underlying native socket.

```
native_handle_type release(
    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. Ownership of the native socket is then transferred to the caller.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

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_context);
...
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_context);
...
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_context);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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 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.

5.72.53 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_context);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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_context);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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::set_option

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);

template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.72.55.1 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_context);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.72.55.2 basic_seq_packet_socket::set_option (2 of 2 overloads)

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void 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_context);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.72.56 basic_seq_packet_socket::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

void shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.72.56.1 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_context);
...
socket.shutdown(asio::ip::tcp::socket::shutdown_send);
```

5.72.56.2 basic_seq_packet_socket::shutdown (2 of 2 overloads)

Inherited from basic_socket.

Disable sends or receives on the socket.

```
void 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_context);
...
asio::error_code ec;
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);
if (ec)
{
    // An error occurred.
}
```

5.72.57 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.

5.72.58 basic_seq_packet_socket::wait

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);  
  
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

5.72.58.1 basic_seq_packet_socket::wait (1 of 2 overloads)

Inherited from basic_socket.

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);
```

This function is used to perform a blocking wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

Example

Waiting for a socket to become readable.

```
asio::ip::tcp::socket socket(io_context);  
...  
socket.wait(asio::ip::tcp::socket::wait_read);
```

5.72.58.2 basic_seq_packet_socket::wait (2 of 2 overloads)

Inherited from basic_socket.

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

This function is used to perform a blocking wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

ec Set to indicate what error occurred, if any.

Example

Waiting for a socket to become readable.

```
asio::ip::tcp::socket socket(io_context);
...
asio::error_code ec;
socket.wait(asio::ip::tcp::socket::wait_read, ec);
```

5.72.59 basic_seq_packet_socket::wait_type

Inherited from `socket_base`.

Wait types.

```
enum wait_type
```

Values

wait_read Wait for a socket to become ready to read.

wait_write Wait for a socket to become ready to write.

wait_error Wait for a socket to have error conditions pending.

For use with `basic_socket::wait()` and `basic_socket::async_wait()`.

5.72.60 basic_seq_packet_socket::~basic_seq_packet_socket

Destroys the socket.

```
~basic_seq_packet_socket();
```

This function destroys the socket, cancelling any outstanding asynchronous operations associated with the socket as if by calling `cancel`.

5.73 basic_socket

Provides socket functionality.

```
template<
    typename Protocol>
class basic_socket :
    public socket_base
```

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.

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.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.

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.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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.
release	Release ownership of the underlying native socket.
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

Protected Member Functions

Name	Description
~basic_socket	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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

5.73.1 basic_socket::assign

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);

void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.73.1.1 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);
```

5.73.1.2 basic_socket::assign (2 of 2 overloads)

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.73.2 basic_socket::async_connect

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
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_context::post()`.

Example

```
void connect_handler(const asio::error_code& error)  
{  
    if (!error)  
    {  
        // Connect succeeded.  
    }  
}  
  
...  
  
asio::ip::tcp::socket socket(io_context);  
asio::ip::tcp::endpoint endpoint(  
    asio::ip::address::from_string("1.2.3.4"), 12345);  
socket.async_connect(endpoint, connect_handler);
```

5.73.3 basic_socket::async_wait

Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
template<  
    typename WaitHandler>  
DEDUCED async_wait(  
    wait_type w,  
    WaitHandler && handler);
```

This function is used to perform an asynchronous wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

handler The handler to be called when the wait 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_context::post()`.

Example

```
void wait_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Wait succeeded.
    }
}

...
asio::ip::tcp::socket socket(io_context);
...
socket.async_wait(asio::ip::tcp::socket::wait_read, wait_handler);
```

5.73.4 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;
```

5.73.4.1 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.

5.73.4.2 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.

5.73.5 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;
```

5.73.5.1 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.

5.73.5.2 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.

5.73.6 basic_socket::basic_socket

Construct a **basic_socket** without opening it.

```
explicit basic_socket(
    asio::io_context & io_context);
```

Construct and open a **basic_socket**.

```
basic_socket(
    asio::io_context & io_context,
    const protocol_type & protocol);
```

Construct a **basic_socket**, opening it and binding it to the given local endpoint.

```
basic_socket(
    asio::io_context & io_context,
    const endpoint_type & endpoint);
```

Construct a **basic_socket** on an existing native socket.

```
basic_socket(
    asio::io_context & io_context,
    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 Protocol>
basic_socket(
    basic_socket< Protocol > && other,
    typename enable_if< is_convertible< Protocol, Protocol >::value >::type * = 0);
```

5.73.6.1 basic_socket::basic_socket (1 of 6 overloads)

Construct a **basic_socket** without opening it.

```
basic_socket(
    asio::io_context & io_context);
```

This constructor creates a socket without opening it.

Parameters

io_context The **io_context** object that the socket will use to dispatch handlers for any asynchronous operations performed on the socket.

5.73.6.2 basic_socket::basic_socket (2 of 6 overloads)

Construct and open a **basic_socket**.

```
basic_socket(
    asio::io_context & io_context,
    const protocol_type & protocol);
```

This constructor creates and opens a socket.

Parameters

io_context The **io_context** 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.

5.73.6.3 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_context & io_context,
    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_context The **io_context** 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.

5.73.6.4 basic_socket::basic_socket (4 of 6 overloads)

Construct a **basic_socket** on an existing native socket.

```
basic_socket(
    asio::io_context & io_context,
    const protocol_type & protocol,
    const native_handle_type & native_socket);
```

This constructor creates a socket object to hold an existing native socket.

Parameters

io_context The `io_context` 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.

5.73.6.5 `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_context&)` constructor.

5.73.6.6 `basic_socket::basic_socket (6 of 6 overloads)`

Move-construct a `basic_socket` from a socket of another protocol type.

```
template<
    typename Protocol1>
basic_socket(
    basic_socket< Protocol1 > && 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_context&)` constructor.

5.73.7 basic_socket::bind

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

void bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.73.7.1 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_context);
socket.open(asio::ip::tcp::v4());
socket.bind(asio::ip::tcp::endpoint(
    asio::ip::tcp::v4(), 12345));
```

5.73.7.2 basic_socket::bind (2 of 2 overloads)

Bind the socket to the given local endpoint.

```
void 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_context);
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.
}
```

5.73.8 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_context);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
asio::socket_base::broadcast option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.73.9 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_context);
...
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

5.73.10 basic_socket::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();

void cancel(
    asio::error_code & ec);
```

5.73.10.1 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.

5.73.10.2 basic_socket::cancel (2 of 2 overloads)

Cancel all asynchronous operations associated with the socket.

```
void 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.

5.73.11 basic_socket::close

Close the socket.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.73.11.1 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.

5.73.11.2 basic_socket::close (2 of 2 overloads)

Close the socket.

```
void 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_context);
...
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.

5.73.12 basic_socket::connect

Connect the socket to the specified endpoint.

```
void connect(
    const endpoint_type & peer_endpoint);

void connect(
    const endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

5.73.12.1 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_context);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.connect(endpoint);
```

5.73.12.2 basic_socket::connect (2 of 2 overloads)

Connect the socket to the specified endpoint.

```
void 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_context);
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.
}
```

5.73.13 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_context);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::debug option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.73.14 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_context);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.73.15 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_context);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.73.16 basic_socket::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.73.17 basic_socket::executor_type

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.73.18 basic_socket::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.73.19 basic_socket::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.73.20 basic_socket::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.73.21 basic_socket::get_option

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;

template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.73.21.1 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_context);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.73.21.2 basic_socket::get_option (2 of 2 overloads)

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void 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_context);
...
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();
```

5.73.22 basic_socket::io_control

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.73.22.1 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_context);
...
asio::ip::tcp::socket::bytes_readable command;
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

5.73.22.2 basic_socket::io_control (2 of 2 overloads)

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void 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_context);
...
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();
```

5.73.23 basic_socket::is_open

Determine whether the socket is open.

```
bool is_open() const;
```

5.73.24 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_context);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.73.25 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_context);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.73.26 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;
```

5.73.26.1 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_context);  
...  
asio::ip::tcp::endpoint endpoint = socket.local_endpoint();
```

5.73.26.2 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_context);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.local_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.73.27 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;
```

5.73.27.1 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.

5.73.27.2 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.

5.73.28 basic_socket::lowest_layer_type

A `basic_socket` is always the lowest layer.

```
typedef basic_socket< Protocol > 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>executor_type</code>	The type of the executor associated with the 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>out_of_band_inline</code>	Socket option for putting received out-of-band data inline.
<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.
<code>send_low_watermark</code>	Socket option for the send low watermark.
<code>shutdown_type</code>	Different ways a socket may be shutdown.
<code>wait_type</code>	Wait types.

Member Functions

Name	Description
<code>assign</code>	Assign an existing native socket to the socket.
<code>async_connect</code>	Start an asynchronous connect.
<code>async_wait</code>	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
<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.
<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_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use <code>get_executor()</code> .) Get the <code>io_context</code> associated with the object.
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code> .) Get the <code>io_context</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_handle</code>	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_socket from another. Move-assign a basic_socket from a socket of another protocol type.
release	Release ownership of the underlying native socket.
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

Protected Member Functions

Name	Description
~basic_socket	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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`

5.73.29 `basic_socket::max_connections`

Inherited from socket_base.

(Deprecated: Use `max_listen_connections`.) The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.73.30 `basic_socket::max_listen_connections`

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_listen_connections = implementation_defined;
```

5.73.31 `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;
```

5.73.32 `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;
```

5.73.33 `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

5.73.34 basic_socket::message_out_of_band

Inherited from socket_base.

Process out-of-band data.

```
static const int message_out_of_band = implementation_defined;
```

5.73.35 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;
```

5.73.36 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.

5.73.37 basic_socket::native_handle_type

The native representation of a socket.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.73.38 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);
```

```
void native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.73.38.1 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, op);
}

```

5.73.38.2 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, op);
}

```

5.73.38.3 basic_socket::native_non_blocking (3 of 3 overloads)

Sets the non-blocking mode of the native socket implementation.

```

void 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, op);
}

```

5.73.39 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);

void non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.73.39.1 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`.

5.73.39.2 `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`.

5.73.39.3 `basic_socket::non_blocking` (3 of 3 overloads)

Sets the non-blocking mode of the socket.

```
void 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`.

5.73.40 basic_socket::open

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());  
  
void open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.73.40.1 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_context);
socket.open(asio::ip::tcp::v4());
```

5.73.40.2 basic_socket::open (2 of 2 overloads)

Open the socket using the specified protocol.

```
void 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_context);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.73.41 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 Protocol1>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_socket >::type & operator=(  
    basic_socket< Protocol1 > && other);
```

5.73.41.1 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_context&)` constructor.

5.73.41.2 basic_socket::operator= (2 of 2 overloads)

Move-assign a **basic_socket** from a socket of another protocol type.

```
template<
    typename Protocol1>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_socket >::type & operator=(  
    basic_socket< Protocol1 > && 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_context&)` constructor.

5.73.42 `basic_socket::out_of_band_inline`

Inherited from socket_base.

Socket option for putting received out-of-band data inline.

```
typedef implementation_defined out_of_band_inline;
```

Implements the SOL_SOCKET/SO_OOBINLINE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option;
socket.get_option(option);
bool value = option.value();
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.73.43 `basic_socket::protocol_type`

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_socket.hpp

Convenience header: asio.hpp

5.73.44 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_context);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.73.45 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_context);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.73.46 basic_socket::release

Release ownership of the underlying native socket.

```
native_handle_type release();  
  
native_handle_type release(  
    asio::error_code & ec);
```

5.73.46.1 basic_socket::release (1 of 2 overloads)

Release ownership of the underlying native socket.

```
native_handle_type release();
```

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. Ownership of the native socket is then transferred to the caller.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

5.73.46.2 basic_socket::release (2 of 2 overloads)

Release ownership of the underlying native socket.

```
native_handle_type release(  
    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. Ownership of the native socket is then transferred to the caller.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

5.73.47 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;
```

5.73.47.1 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_context);  
...  
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint();
```

5.73.47.2 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_context);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.73.48 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_context);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.73.49 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_context);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.73.50 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_context);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.73.51 basic_socket::set_option

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);

template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.73.51.1 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_context);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.73.51.2 basic_socket::set_option (2 of 2 overloads)

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void 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_context);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.73.52 basic_socket::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

void shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.73.52.1 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_context);
...
socket.shutdown(asio::ip::socket::shutdown_send);
```

5.73.52.2 basic_socket::shutdown (2 of 2 overloads)

Disable sends or receives on the socket.

```
void 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_context);
...
asio::error_code ec;
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);
if (ec)
{
    // An error occurred.
}
```

5.73.53 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.

5.73.54 basic_socket::wait

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(
    wait_type w);

void wait(
    wait_type w,
    asio::error_code & ec);
```

5.73.54.1 basic_socket::wait (1 of 2 overloads)

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);
```

This function is used to perform a blocking wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

Example

Waiting for a socket to become readable.

```
asio::ip::tcp::socket socket(io_context);  
...  
socket.wait(asio::ip::tcp::socket::wait_read);
```

5.73.54.2 basic_socket::wait (2 of 2 overloads)

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

This function is used to perform a blocking wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

ec Set to indicate what error occurred, if any.

Example

Waiting for a socket to become readable.

```
asio::ip::tcp::socket socket(io_context);  
...  
asio::error_code ec;  
socket.wait(asio::ip::tcp::socket::wait_read, ec);
```

5.73.55 basic_socket::wait_type

Inherited from socket_base.

Wait types.

```
enum wait_type
```

Values

wait_read Wait for a socket to become ready to read.

wait_write Wait for a socket to become ready to write.

wait_error Wait for a socket to have error conditions pending.

For use with `basic_socket::wait()` and `basic_socket::async_wait()`.

5.73.56 basic_socket::~basic_socket

Protected destructor to prevent deletion through this type.

```
~basic_socket();
```

This function destroys the socket, cancelling any outstanding asynchronous operations associated with the socket as if by calling `cancel`.

5.74 basic_socket_acceptor

Provides the ability to accept new connections.

```
template<
    typename Protocol>
class basic_socket_acceptor :
    public socket_base
```

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.
executor_type	The type of the executor associated with the 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.

Name	Description
native_handle_type	The native representation of an acceptor.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.
async_wait	Asynchronously wait for the acceptor to become ready to read, ready to write, or to have pending error conditions.
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_executor	Get the executor associated with the object.

Name	Description
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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.
release	Release ownership of the underlying native acceptor.
set_option	Set an option on the acceptor.
wait	Wait for the acceptor to become ready to read, ready to write, or to have pending error conditions.
~basic_socket_acceptor	Destroys the acceptor.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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_context);
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`

5.74.1 basic_socket_acceptor::accept

Accept a new connection.

```
template<
    typename Protocol1>
void accept(
    basic_socket< Protocol1 > & peer,
    typename enable_if< is_convertible< Protocol, Protocol1 >::value >::type * = 0);

template<
    typename Protocol1>
void accept(
    basic_socket< Protocol1 > & 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.

```

void accept(
    basic_socket< protocol_type > & peer,
    endpoint_type & peer_endpoint);

void accept(
    basic_socket< protocol_type > & peer,
    endpoint_type & peer_endpoint,
    asio::error_code & ec);

Protocol::socket accept();

Protocol::socket accept(
    asio::error_code & ec);

Protocol::socket accept(
    asio::io_context & io_context);

Protocol::socket accept(
    asio::io_context & io_context,
    asio::error_code & ec);

Protocol::socket accept(
    endpoint_type & peer_endpoint);

Protocol::socket accept(
    endpoint_type & peer_endpoint,
    asio::error_code & ec);

Protocol::socket accept(
    asio::io_context & io_context,
    endpoint_type & peer_endpoint);

Protocol::socket accept(
    asio::io_context & io_context,
    endpoint_type & peer_endpoint,
    asio::error_code & ec);

```

5.74.1.1 basic_socket_acceptor::accept (1 of 12 overloads)

Accept a new connection.

```

template<
    typename Protocol1>
void accept(
    basic_socket< Protocol1 > & 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_context);
...
asio::ip::tcp::socket socket(io_context);
acceptor.accept(socket);
```

5.74.1.2 basic_socket_acceptor::accept (2 of 12 overloads)

Accept a new connection.

```
template<
    typename Protocol1>
void accept(
    basic_socket< Protocol1 > & 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_context);
...
asio::ip::tcp::socket socket(io_context);
asio::error_code ec;
acceptor.accept(socket, ec);
if (ec)
{
    // An error occurred.
}
```

5.74.1.3 basic_socket_acceptor::accept (3 of 12 overloads)

Accept a new connection and obtain the endpoint of the peer.

```
void accept(
    basic_socket< protocol_type > & 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_context);
...
asio::ip::tcp::socket socket(io_context);
asio::ip::tcp::endpoint endpoint;
acceptor.accept(socket, endpoint);
```

5.74.1.4 basic_socket_acceptor::accept (4 of 12 overloads)

Accept a new connection and obtain the endpoint of the peer.

```
void accept(
    basic_socket< protocol_type > & 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_context);
...
asio::ip::tcp::socket socket(io_context);
asio::ip::tcp::endpoint endpoint;
asio::error_code ec;
acceptor.accept(socket, endpoint, ec);
if (ec)
{
    // An error occurred.
}
```

5.74.1.5 basic_socket_acceptor::accept (5 of 12 overloads)

Accept a new connection.

```
Protocol::socket accept();
```

This function is used to accept a new connection from a peer. The function call will block until a new connection has been accepted successfully or an error occurs.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Return Value

A socket object representing the newly accepted connection.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::socket socket(acceptor.accept());
```

5.74.1.6 basic_socket_acceptor::accept (6 of 12 overloads)

Accept a new connection.

```
Protocol::socket accept(
    asio::error_code & ec);
```

This function is used to accept a new connection from a peer. The function call will block until a new connection has been accepted successfully or an error occurs.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

On success, a socket object representing the newly accepted connection. On error, a socket object where `is_open()` is false.

Example

```
asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::socket socket(acceptor.accept(ec));
if (ec)
{
    // An error occurred.
}
```

5.74.1.7 basic_socket_acceptor::accept (7 of 12 overloads)

Accept a new connection.

```
Protocol::socket accept(
   asio::io_context & io_context);
```

This function is used to accept a new connection from a peer. The function call will block until a new connection has been accepted successfully or an error occurs.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Parameters

io_context The `io_context` object to be used for the newly accepted socket.

Return Value

A socket object representing the newly accepted connection.

Exceptions

`asio::system_error` Thrown on failure.

Example

```
asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::socket socket(acceptor.accept());
```

5.74.1.8 basic_socket_acceptor::accept (8 of 12 overloads)

Accept a new connection.

```
Protocol::socket accept(
    asio::io_context & io_context,
    asio::error_code & ec);
```

This function is used to accept a new connection from a peer. The function call will block until a new connection has been accepted successfully or an error occurs.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Parameters

io_context The `io_context` object to be used for the newly accepted socket.

ec Set to indicate what error occurred, if any.

Return Value

On success, a socket object representing the newly accepted connection. On error, a socket object where `is_open()` is false.

Example

```
asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::socket socket(acceptor.accept(io_context2, ec));
if (ec)
{
    // An error occurred.
}
```

5.74.1.9 basic_socket_acceptor::accept (9 of 12 overloads)

Accept a new connection.

```
Protocol::socket accept(
    endpoint_type & peer_endpoint);
```

This function is used to accept a new connection from a peer. The function call will block until a new connection has been accepted successfully or an error occurs.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Parameters

peer_endpoint An endpoint object into which the endpoint of the remote peer will be written.

Return Value

A socket object representing the newly accepted connection.

Exceptions

asio::system_error Thrown on failure.

Example

```
asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::endpoint endpoint;
asio::ip::tcp::socket socket(acceptor.accept(endpoint));
```

5.74.1.10 basic_socket_acceptor::accept (10 of 12 overloads)

Accept a new connection.

```
Protocol::socket accept(
    endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

This function is used to accept a new connection from a peer. The function call will block until a new connection has been accepted successfully or an error occurs.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Parameters

peer_endpoint An endpoint object into which the endpoint of the remote peer will be written.

ec Set to indicate what error occurred, if any.

Return Value

On success, a socket object representing the newly accepted connection. On error, a socket object where `is_open()` is false.

Example

```
asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::endpoint endpoint;
asio::ip::tcp::socket socket(acceptor.accept(endpoint, ec));
if (ec)
{
    // An error occurred.
}
```

5.74.1.11 basic_socket_acceptor::accept (11 of 12 overloads)

Accept a new connection.

```
Protocol::socket accept(
    asio::io_context & io_context,
    endpoint_type & peer_endpoint);
```

This function is used to accept a new connection from a peer. The function call will block until a new connection has been accepted successfully or an error occurs.

This overload requires that the `Protocol` template parameter satisfy the `AcceptableProtocol` type requirements.

Parameters

io_context The `io_context` object to be used for the newly accepted socket.

peer_endpoint An endpoint object into which the endpoint of the remote peer will be written.

Return Value

A socket object representing the newly accepted connection.

Exceptions

`asio::system_error` Thrown on failure.

Example

```
asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::endpoint endpoint;
asio::ip::tcp::socket socket(
    acceptor.accept(io_context2, endpoint));
```

5.74.1.12 basic_socket_acceptor::accept (12 of 12 overloads)

Accept a new connection.

```
Protocol::socket accept(
    asio::io_context & io_context,
    endpoint_type & peer_endpoint,
    asio::error_code & ec);
```

This function is used to accept a new connection from a peer. The function call will block until a new connection has been accepted successfully or an error occurs.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Parameters

io_context The `io_context` object to be used for the newly accepted socket.

peer_endpoint An endpoint object into which the endpoint of the remote peer will be written.

ec Set to indicate what error occurred, if any.

Return Value

On success, a socket object representing the newly accepted connection. On error, a socket object where `is_open()` is false.

Example

```
asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::endpoint endpoint;
asio::ip::tcp::socket socket(
    acceptor.accept(io_context2, endpoint, ec));
if (ec)
{
    // An error occurred.
}
```

5.74.2 basic_socket_acceptor::assign

Assigns an existing native acceptor to the acceptor.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_acceptor);

void assign(
    const protocol_type & protocol,
    const native_handle_type & native_acceptor,
    asio::error_code & ec);
```

5.74.2.1 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);
```

5.74.2.2 basic_socket_acceptor::assign (2 of 2 overloads)

Assigns an existing native acceptor to the acceptor.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_acceptor,
    asio::error_code & ec);
```

5.74.3 basic_socket_acceptor::async_accept

Start an asynchronous accept.

```
template<
    typename Protocol1,
    typename AcceptHandler>
DEDUCED async_accept(
    basic_socket< Protocol1 > & peer,
    AcceptHandler && handler,
    typename enable_if< is_convertible< Protocol, Protocol1 >::value >::type * = 0);

template<
    typename AcceptHandler>
DEDUCED async_accept(
    basic_socket< protocol_type > & peer,
    endpoint_type & peer_endpoint,
    AcceptHandler && handler);

template<
    typename MoveAcceptHandler>
DEDUCED async_accept(
    MoveAcceptHandler && handler);

template<
    typename MoveAcceptHandler>
DEDUCED async_accept(
    asio::io_context & io_context,
    MoveAcceptHandler && handler);

template<
    typename MoveAcceptHandler>
DEDUCED async_accept(
    endpoint_type & peer_endpoint,
    MoveAcceptHandler && handler);

template<
    typename MoveAcceptHandler>
```

```
DEDUCED async_accept(
    asio::io_context & io_context,
    endpoint_type & peer_endpoint,
    MoveAcceptHandler && handler);
```

5.74.3.1 basic_socket_acceptor::async_accept (1 of 6 overloads)

Start an asynchronous accept.

```
template<
    typename Protocol1,
    typename AcceptHandler>
DEDUCED async_accept(
    basic_socket< Protocol1 > & 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_context::post()`.

Example

```
void accept_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Accept succeeded.
    }
}

...

asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::socket socket(io_context);
acceptor.async_accept(socket, accept_handler);
```

5.74.3.2 basic_socket_acceptor::async_accept (2 of 6 overloads)

Start an asynchronous accept.

```
template<
    typename AcceptHandler>
DEDUCED async_accept(
    basic_socket< protocol_type > & 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_context::post()`.

5.74.3.3 basic_socket_acceptor::async_accept (3 of 6 overloads)

Start an asynchronous accept.

```
template<
    typename MoveAcceptHandler>
DEDUCED async_accept(
    MoveAcceptHandler && handler);
```

This function is used to asynchronously accept a new connection. The function call always returns immediately.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Parameters

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.
    typename Protocol::socket peer // On success, the newly accepted socket.
);
```

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_context::post()`.

Example

```
void accept_handler(const asio::error_code& error,
    asio::ip::tcp::socket peer)
{
    if (!error)
    {
        // Accept succeeded.
    }
}

...
asio::ip::tcp::acceptor acceptor(io_context);
...
acceptor.async_accept(accept_handler);
```

5.74.3.4 basic_socket_acceptor::async_accept (4 of 6 overloads)

Start an asynchronous accept.

```
template<
    typename MoveAcceptHandler>
DEDUCED async_accept(
    asio::io_context & io_context,
    MoveAcceptHandler && handler);
```

This function is used to asynchronously accept a new connection. The function call always returns immediately.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Parameters

io_context The `io_context` object to be used for the newly accepted socket.

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.
    typename Protocol::socket peer // On success, the newly accepted socket.
);
```

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_context::post()`.

Example

```
void accept_handler(const asio::error_code& error,
    asio::ip::tcp::socket peer)
{
    if (!error)
    {
        // Accept succeeded.
    }
}

...
```

```
asio::ip::tcp::acceptor acceptor(io_context);
...
acceptor.async_accept(io_context2, accept_handler);
```

5.74.3.5 basic_socket_acceptor::async_accept (5 of 6 overloads)

Start an asynchronous accept.

```
template<
    typename MoveAcceptHandler>
DEDUCED async_accept(
    endpoint_type & peer_endpoint,
    MoveAcceptHandler && handler);
```

This function is used to asynchronously accept a new connection. The function call always returns immediately.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Parameters

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.
    typename Protocol::socket peer // On success, the newly accepted socket.
);
```

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_context::post()`.

Example

```
void accept_handler(const asio::error_code& error,
    asio::ip::tcp::socket peer)
{
    if (!error)
    {
        // Accept succeeded.
    }
}

...

asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::endpoint endpoint;
acceptor.async_accept(endpoint, accept_handler);
```

5.74.3.6 basic_socket_acceptor::async_accept (6 of 6 overloads)

Start an asynchronous accept.

```
template<
    typename MoveAcceptHandler>
DEDUCED async_accept(
    asio::io_context & io_context,
    endpoint_type & peer_endpoint,
    MoveAcceptHandler && handler);
```

This function is used to asynchronously accept a new connection. The function call always returns immediately.

This overload requires that the Protocol template parameter satisfy the AcceptableProtocol type requirements.

Parameters

io_context The `io_context` object to be used for the newly accepted socket.

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.
    typename Protocol::socket peer // On success, the newly accepted socket.
);
```

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_context::post()`.

Example

```
void accept_handler(const asio::error_code& error,
    asio::ip::tcp::socket peer)
{
    if (!error)
    {
        // Accept succeeded.
    }
}

...

asio::ip::tcp::acceptor acceptor(io_context);
...
asio::ip::tcp::endpoint endpoint;
acceptor.async_accept(io_context2, endpoint, accept_handler);
```

5.74.4 basic_socket_acceptor::async_wait

Asynchronously wait for the acceptor to become ready to read, ready to write, or to have pending error conditions.

```
template<
    typename WaitHandler>
DEDUCED async_wait(
    wait_type w,
    WaitHandler && handler);
```

This function is used to perform an asynchronous wait for an acceptor to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired acceptor state.

handler The handler to be called when the wait 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_context::post()`.

Example

```
void wait_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Wait succeeded.
    }
}

...

asio::ip::tcp::acceptor acceptor(io_context);
...
acceptor.async_wait(
    asio::ip::tcp::acceptor::wait_read,
    wait_handler);
```

5.74.5 basic_socket_acceptor::basic_socket_acceptor

Construct an acceptor without opening it.

```
explicit basic_socket_acceptor(
    asio::io_context & io_context);
```

Construct an open acceptor.

```
basic_socket_acceptor(
    asio::io_context & io_context,
    const protocol_type & protocol);
```

Construct an acceptor opened on the given endpoint.

```
basic_socket_acceptor(
    asio::io_context & io_context,
    const endpoint_type & endpoint,
    bool reuse_addr = true);
```

Construct a `basic_socket_acceptor` on an existing native acceptor.

```
basic_socket_acceptor(
   asio::io_context & io_context,
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>
basic_socket_acceptor(
    basic_socket_acceptor< Protocol1 > && other,
typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.74.5.1 `basic_socket_acceptor::basic_socket_acceptor (1 of 6 overloads)`

Construct an acceptor without opening it.

```
basic_socket_acceptor(
    asio::io_context & io_context);
```

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_context The `io_context` object that the acceptor will use to dispatch handlers for any asynchronous operations performed on the acceptor.

5.74.5.2 `basic_socket_acceptor::basic_socket_acceptor (2 of 6 overloads)`

Construct an open acceptor.

```
basic_socket_acceptor(
    asio::io_context & io_context,
const protocol_type & protocol);
```

This constructor creates an acceptor and automatically opens it.

Parameters

io_context The `io_context` 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.

5.74.5.3 basic_socket_acceptor::basic_socket_acceptor (3 of 6 overloads)

Construct an acceptor opened on the given endpoint.

```
basic_socket_acceptor(
    asio::io_context & io_context,
    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_context The `io_context` 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_context);
acceptor.open(endpoint.protocol());
if (reuse_addr)
    acceptor.set_option(socket_base::reuse_address(true));
acceptor.bind(endpoint);
acceptor.listen(listen_backlog);
```

5.74.5.4 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_context & io_context,
    const protocol_type & protocol,
    const native_handle_type & native_acceptor);
```

This constructor creates an acceptor object to hold an existing native acceptor.

Parameters

io_context The `io_context` 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.

5.74.5.5 `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_context&)` constructor.

5.74.5.6 `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>
basic_socket_acceptor(
    basic_socket_acceptor< Protocol1 > && 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_context&)` constructor.

5.74.6 `basic_socket_acceptor::bind`

Bind the acceptor to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

void bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.74.6.1 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_context);
asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), 12345);
acceptor.open(endpoint.protocol());
acceptor.bind(endpoint);
```

5.74.6.2 basic_socket_acceptor::bind (2 of 2 overloads)

Bind the acceptor to the given local endpoint.

```
void 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_context);
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.
}
```

5.74.7 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_context);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.74.8 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_context);
...
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

5.74.9 basic_socket_acceptor::cancel

Cancel all asynchronous operations associated with the acceptor.

```
void cancel();  
  
void cancel(  
    asio::error_code & ec);
```

5.74.9.1 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.

5.74.9.2 basic_socket_acceptor::cancel (2 of 2 overloads)

Cancel all asynchronous operations associated with the acceptor.

```
void 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.

5.74.10 basic_socket_acceptor::close

Close the acceptor.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.74.10.1 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.

5.74.10.2 basic_socket_acceptor::close (2 of 2 overloads)

Close the acceptor.

```
void 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_context);  
...  
asio::error_code ec;  
acceptor.close(ec);  
if (ec)  
{  
    // An error occurred.  
}
```

5.74.11 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_context);  
...  
asio::socket_base::debug option(true);  
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);  
...  
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

5.74.12 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_context);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.74.13 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_context);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.74.14 basic_socket_acceptor::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.74.15 basic_socket_acceptor::executor_type

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.

Name	Description
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.74.16 basic_socket_acceptor::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.74.17 basic_socket_acceptor::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.74.18 basic_socket_acceptor::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.74.19 `basic_socket_acceptor::get_option`

Get an option from the acceptor.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option);

template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option,
    asio::error_code & ec);
```

5.74.19.1 `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_context);
...
asio::ip::tcp::acceptor::reuse_address option;
acceptor.get_option(option);
bool is_set = option.get();
```

5.74.19.2 basic_socket_acceptor::get_option (2 of 2 overloads)

Get an option from the acceptor.

```
template<
    typename GettableSocketOption>
void 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_context);
...
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();
```

5.74.20 basic_socket_acceptor::io_control

Perform an IO control command on the acceptor.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.74.20.1 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_context);
...
asio::ip::tcp::acceptor::non_blocking_io command(true);
socket.io_control(command);
```

5.74.20.2 basic_socket_acceptor::io_control (2 of 2 overloads)

Perform an IO control command on the acceptor.

```
template<
    typename IoControlCommand>
void 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_context);
...
asio::ip::tcp::acceptor::non_blocking_io command(true);
asio::error_code ec;
socket.io_control(command, ec);
if (ec)
{
    // An error occurred.
}
```

5.74.21 basic_socket_acceptor::is_open

Determine whether the acceptor is open.

```
bool is_open() const;
```

5.74.22 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_context);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.74.23 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_context);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.74.24 basic_socket_acceptor::listen

Place the acceptor into the state where it will listen for new connections.

```
void listen(  
    int backlog = socket_base::max_listen_connections);  
  
void listen(  
    int backlog,  
    asio::error_code & ec);
```

5.74.24.1 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_listen_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.

5.74.24.2 basic_socket_acceptor::listen (2 of 2 overloads)

Place the acceptor into the state where it will listen for new connections.

```
void 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_context);
...
asio::error_code ec;
acceptor.listen(asio::socket_base::max_listen_connections, ec);
if (ec)
{
    // An error occurred.
}
```

5.74.25 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;
```

5.74.25.1 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_context);
...
asio::ip::tcp::endpoint endpoint = acceptor.local_endpoint();
```

5.74.25.2 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_context);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = acceptor.local_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.74.26 basic_socket_acceptor::max_connections

Inherited from socket_base.

(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.74.27 basic_socket_acceptor::max_listen_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_listen_connections = implementation_defined;
```

5.74.28 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;
```

5.74.29 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;
```

5.74.30 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

5.74.31 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;
```

5.74.32 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;
```

5.74.33 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.

5.74.34 basic_socket_acceptor::native_handle_type

The native representation of an acceptor.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.74.35 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);
```

```
void native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.74.35.1 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.

5.74.35.2 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.

5.74.35.3 basic_socket_acceptor::native_non_blocking (3 of 3 overloads)

Sets the non-blocking mode of the native acceptor implementation.

```
void 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.

5.74.36 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);

void non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.74.36.1 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`.

5.74.36.2 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`.

5.74.36.3 basic_socket_acceptor::non_blocking (3 of 3 overloads)

Sets the non-blocking mode of the acceptor.

```
void 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`.

5.74.37 basic_socket_acceptor::open

Open the acceptor using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());
```



```
void open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.74.37.1 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_context);
acceptor.open(asio::ip::tcp::v4());
```

5.74.37.2 basic_socket_acceptor::open (2 of 2 overloads)

Open the acceptor using the specified protocol.

```
void 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_context);
asio::error_code ec;
acceptor.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.74.38 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>  
enable_if< is_convertible< Protocol1, Protocol >::value, basic_socket_acceptor >::type & ←  
operator=(  
    basic_socket_acceptor< Protocol1 > && other);
```

5.74.38.1 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_context&)` constructor.

5.74.38.2 basic_socket_acceptor::operator= (2 of 2 overloads)

Move-assign a `basic_socket_acceptor` from an acceptor of another protocol type.

```
template<  
    typename Protocol1>  
enable_if< is_convertible< Protocol1, Protocol >::value, basic_socket_acceptor >::type & ←  
operator=(  
    basic_socket_acceptor< Protocol1 > && 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_context&)` constructor.

5.74.39 basic_socket_acceptor::out_of_band_inline

Inherited from socket_base.

Socket option for putting received out-of-band data inline.

```
typedef implementation_defined out_of_band_inline;
```

Implements the SOL_SOCKET/SO_OOBINLINE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option;
socket.get_option(option);
bool value = option.value();
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.74.40 basic_socket_acceptor::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_socket_acceptor.hpp

Convenience header: asio.hpp

5.74.41 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_context);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.74.42 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_context);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.74.43 basic_socket_acceptor::release

Release ownership of the underlying native acceptor.

```
native_handle_type release();  
  
native_handle_type release(  
    asio::error_code & ec);
```

5.74.43.1 basic_socket_acceptor::release (1 of 2 overloads)

Release ownership of the underlying native acceptor.

```
native_handle_type release();
```

This function causes all outstanding asynchronous accept operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error. Ownership of the native acceptor is then transferred to the caller.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

5.74.43.2 basic_socket_acceptor::release (2 of 2 overloads)

Release ownership of the underlying native acceptor.

```
native_handle_type release(  
    asio::error_code & ec);
```

This function causes all outstanding asynchronous accept operations to finish immediately, and the handlers for cancelled operations will be passed the `asio::error::operation_aborted` error. Ownership of the native acceptor is then transferred to the caller.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

5.74.44 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_context);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.74.45 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_context);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.74.46 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_context);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.74.47 basic_socket_acceptor::set_option

Set an option on the acceptor.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);

template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.74.47.1 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_context);
...
asio::ip::tcp::acceptor::reuse_address option(true);
acceptor.set_option(option);
```

5.74.47.2 basic_socket_acceptor::set_option (2 of 2 overloads)

Set an option on the acceptor.

```
template<
    typename SettableSocketOption>
void 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_context);
...
asio::ip::tcp::acceptor::reuse_address option(true);
asio::error_code ec;
acceptor.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.74.48 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.

5.74.49 basic_socket_acceptor::wait

Wait for the acceptor to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);  
  
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

5.74.49.1 basic_socket_acceptor::wait (1 of 2 overloads)

Wait for the acceptor to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);
```

This function is used to perform a blocking wait for an acceptor to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired acceptor state.

Example

Waiting for an acceptor to become readable.

```
asio::ip::tcp::acceptor acceptor(io_context);  
...  
acceptor.wait(asio::ip::tcp::acceptor::wait_read);
```

5.74.49.2 basic_socket_acceptor::wait (2 of 2 overloads)

Wait for the acceptor to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

This function is used to perform a blocking wait for an acceptor to enter a ready to read, write or error condition state.

Parameters

- w Specifies the desired acceptor state.
- ec Set to indicate what error occurred, if any.

Example

Waiting for an acceptor to become readable.

```
asio::ip::tcp::acceptor acceptor(io_context);
...
asio::error_code ec;
acceptor.wait(asio::ip::tcp::acceptor::wait_read, ec);
```

5.74.50 basic_socket_acceptor::wait_type

Inherited from `socket_base`.

Wait types.

```
enum wait_type
```

Values

wait_read Wait for a socket to become ready to read.

wait_write Wait for a socket to become ready to write.

wait_error Wait for a socket to have error conditions pending.

For use with `basic_socket::wait()` and `basic_socket::async_wait()`.

5.74.51 basic_socket_acceptor::~basic_socket_acceptor

Destroys the acceptor.

```
~basic_socket_acceptor();
```

This function destroys the acceptor, cancelling any outstanding asynchronous operations associated with the acceptor as if by calling `cancel`.

5.75 basic_socket_iostream

Iostream interface for a socket.

```
template<
    typename Protocol,
    typename Clock = chrono::steady_clock,
    typename WaitTraits = wait_traits<Clock>>
class basic_socket_iostream
```

Types

Name	Description
clock_type	The clock type.
duration	The duration type.
duration_type	(Deprecated: Use duration.) The duration type.
endpoint_type	The endpoint type.
protocol_type	The protocol type.
time_point	The time type.
time_type	(Deprecated: Use time_point.) The time type.

Member Functions

Name	Description
basic_socket_iostream	Construct a basic_socket_iostream without establishing a connection. Construct a basic_socket_iostream from the supplied socket. Move-construct a basic_socket_iostream from another. 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_after	Set the stream's expiry time relative to now.
expires_at	(Deprecated: Use expiry().) Get the stream's expiry time as an absolute time. Set the stream's expiry time as an absolute time.
expires_from_now	(Deprecated: Use expiry().) Get the stream's expiry time relative to now. (Deprecated: Use expires_after().) Set the stream's expiry time relative to now.
expiry	Get the stream's expiry time as an absolute time.
operator=	Move-assign a basic_socket_iostream from another.
rdbuf	Return a pointer to the underlying streambuf.
socket	Get a reference to the underlying socket.

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.75.1 basic_socket_iostream::basic_socket_iostream

Construct a `basic_socket_iostream` without establishing a connection.

```
basic_socket_iostream();
```

Construct a `basic_socket_iostream` from the supplied socket.

```
explicit basic_socket_iostream(
    basic_stream_socket< protocol_type > s);
```

Move-construct a `basic_socket_iostream` from another.

```
basic_socket_iostream(
    basic_socket_iostream && other);
```

Establish a connection to an endpoint corresponding to a resolver query.

```
template<
    typename T1,
    ... ,
    typename TN>
explicit basic_socket_iostream(
    T1 t1,
    ... ,
    TN tn);
```

5.75.1.1 basic_socket_iostream::basic_socket_iostream (1 of 4 overloads)

Construct a `basic_socket_iostream` without establishing a connection.

```
basic_socket_iostream();
```

5.75.1.2 basic_socket_iostream::basic_socket_iostream (2 of 4 overloads)

Construct a `basic_socket_iostream` from the supplied socket.

```
basic_socket_iostream(
    basic_stream_socket< protocol_type > s);
```

5.75.1.3 basic_socket_iostream::basic_socket_iostream (3 of 4 overloads)

Move-construct a `basic_socket_iostream` from another.

```
basic_socket_iostream(
    basic_socket_iostream && other);
```

5.75.1.4 basic_socket_iostream::basic_socket_iostream (4 of 4 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.

5.75.2 basic_socket_iostream::clock_type

The clock type.

```
typedef Clock clock_type;
```

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.75.3 basic_socket_iostream::close

Close the connection.

```
void close();
```

5.75.4 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.

5.75.5 basic_socket_iostream::duration

The duration type.

```
typedef WaitTraits::duration duration;
```

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.75.6 basic_socket_iostream::duration_type

(Deprecated: Use duration.) The duration type.

```
typedef WaitTraits::duration_type duration_type;
```

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.75.7 basic_socket_iostream::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.75.8 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;
}
```

5.75.9 basic_socket_iostream::expires_after

Set the stream's expiry time relative to now.

```
void expires_after(  
    const duration & 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.

5.75.10 basic_socket_iostream::expires_at

(Deprecated: Use `expiry()`.) Get the stream's expiry time as an absolute time.

```
time_point expires_at() const;
```

Set the stream's expiry time as an absolute time.

```
void expires_at(  
    const time_point & expiry_time);
```

5.75.10.1 basic_socket_iostream::expires_at (1 of 2 overloads)

(Deprecated: Use `expiry()`.) Get the stream's expiry time as an absolute time.

```
time_point expires_at() const;
```

Return Value

An absolute time value representing the stream's expiry time.

5.75.10.2 basic_socket_iostream::expires_at (2 of 2 overloads)

Set the stream's expiry time as an absolute time.

```
void expires_at(  
    const time_point & 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.

5.75.11 basic_socket_iostream::expires_from_now

(Deprecated: Use `expiry()`) Get the stream's expiry time relative to now.

```
duration expires_from_now() const;
```

(Deprecated: Use `expires_after()`) Set the stream's expiry time relative to now.

```
void expires_from_now(
    const duration & expiry_time);
```

5.75.11.1 basic_socket_iostream::expires_from_now (1 of 2 overloads)

(Deprecated: Use `expiry()`) Get the stream's expiry time relative to now.

```
duration expires_from_now() const;
```

Return Value

A relative time value representing the stream's expiry time.

5.75.11.2 basic_socket_iostream::expires_from_now (2 of 2 overloads)

(Deprecated: Use `expires_after()`) Set the stream's expiry time relative to now.

```
void expires_from_now(
    const duration & 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.

5.75.12 basic_socket_iostream::expiry

Get the stream's expiry time as an absolute time.

```
time_point expiry() const;
```

Return Value

An absolute time value representing the stream's expiry time.

5.75.13 basic_socket_iostream::operator=

Move-assign a `basic_socket_iostream` from another.

```
basic_socket_iostream & operator=
    (basic_socket_iostream && other);
```

5.75.14 basic_socket_iostream::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.75.15 basic_socket_iostream::rdbuf

Return a pointer to the underlying streambuf.

```
basic_socket_streambuf< Protocol, Clock, WaitTraits > * rdbuf() const;
```

5.75.16 basic_socket_iostream::socket

Get a reference to the underlying socket.

```
basic_socket< Protocol > & socket();
```

5.75.17 basic_socket_iostream::time_point

The time type.

```
typedef WaitTraits::time_point time_point;
```

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.75.18 basic_socket_iostream::time_type

(Deprecated: Use time_point.) The time type.

```
typedef WaitTraits::time_type time_type;
```

Requirements

Header: asio/basic_socket_iostream.hpp

Convenience header: asio.hpp

5.76 basic_socket_streambuf

Iostream streambuf for a socket.

```
template<
    typename Protocol,
    typename Clock = chrono::steady_clock,
    typename WaitTraits = wait_traits<Clock>>
class basic_socket_streambuf :
    basic_socket< Protocol >
```

Types

Name	Description
clock_type	The clock type.
duration	The duration type.
duration_type	(Deprecated: Use duration.) The duration type.
endpoint_type	The endpoint type.
protocol_type	The protocol type.
time_point	The time type.
time_type	(Deprecated: Use time_point.) The time type.

Member Functions

Name	Description
basic_socket_streambuf	Construct a basic_socket_streambuf without establishing a connection. Construct a basic_socket_streambuf from the supplied socket. Move-construct a basic_socket_streambuf from another.
close	Close the connection.
connect	Establish a connection.
error	Get the last error associated with the stream buffer.
expires_after	Set the stream buffer's expiry time relative to now.
expires_at	(Deprecated: Use expiry().) Get the stream buffer's expiry time as an absolute time. Set the stream buffer's expiry time as an absolute time.

Name	Description
<code>expires_from_now</code>	(Deprecated: Use <code>expiry()</code> .) Get the stream buffer's expiry time relative to now. (Deprecated: Use <code>expires_after()</code> .) Set the stream buffer's expiry time relative to now.
<code>expiry</code>	Get the stream buffer's expiry time as an absolute time.
<code>operator=</code>	Move-assign a <code>basic_socket_streambuf</code> from another.
<code>puberror</code>	(Deprecated: Use <code>error()</code> .) Get the last error associated with the stream buffer.
<code>socket</code>	Get a reference to the underlying socket.
<code>~basic_socket_streambuf</code>	Destructor flushes buffered data.

Protected Member Functions

Name	Description
<code>overflow</code>	
<code>setbuf</code>	
<code>sync</code>	
<code>underflow</code>	

Requirements

Header: `asio/basic_socket_streambuf.hpp`

Convenience header: `asio.hpp`

5.76.1 `basic_socket_streambuf::basic_socket_streambuf`

Construct a `basic_socket_streambuf` without establishing a connection.

```
basic_socket_streambuf();
```

Construct a `basic_socket_streambuf` from the supplied socket.

```
explicit basic_socket_streambuf(
    basic_stream_socket< protocol_type > s);
```

Move-construct a `basic_socket_streambuf` from another.

```
basic_socket_streambuf(
    basic_socket_streambuf && other);
```

5.76.1.1 `basic_socket_streambuf::basic_socket_streambuf (1 of 3 overloads)`

Construct a `basic_socket_streambuf` without establishing a connection.

```
basic_socket_streambuf();
```

5.76.1.2 `basic_socket_streambuf::basic_socket_streambuf (2 of 3 overloads)`

Construct a `basic_socket_streambuf` from the supplied socket.

```
basic_socket_streambuf(  
    basic_stream_socket< protocol_type > s);
```

5.76.1.3 `basic_socket_streambuf::basic_socket_streambuf (3 of 3 overloads)`

Move-construct a `basic_socket_streambuf` from another.

```
basic_socket_streambuf(  
    basic_socket_streambuf && other);
```

5.76.2 `basic_socket_streambuf::clock_type`

The clock type.

```
typedef Clock clock_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.76.3 `basic_socket_streambuf::close`

Close the connection.

```
basic_socket_streambuf * close();
```

Return Value

`this` if a connection was successfully established, a null pointer otherwise.

5.76.4 `basic_socket_streambuf::connect`

Establish a connection.

```
basic_socket_streambuf * connect(  
    const endpoint_type & endpoint);
```

```
template<  
    typename T1,  
    ...,  
    typename TN>  
basic_socket_streambuf * connect(  
    T1 t1,  
    ...,  
    TN tn);
```

5.76.4.1 basic_socket_streambuf::connect (1 of 2 overloads)

Establish a connection.

```
basic_socket_streambuf * 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.

5.76.4.2 basic_socket_streambuf::connect (2 of 2 overloads)

Establish a connection.

```
template<
    typename T1,
    ... ,
    typename TN>
basic_socket_streambuf * 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.

5.76.5 basic_socket_streambuf::duration

The duration type.

```
typedef WaitTraits::duration duration;
```

Requirements

Header: `asio/basic_socket_streambuf.hpp`

Convenience header: `asio.hpp`

5.76.6 basic_socket_streambuf::duration_type

(Deprecated: Use `duration`.) The duration type.

```
typedef WaitTraits::duration_type duration_type;
```

Requirements

Header: `asio/basic_socket_streambuf.hpp`

Convenience header: `asio.hpp`

5.76.7 `basic_socket_streambuf::endpoint_type`

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: `asio/basic_socket_streambuf.hpp`

Convenience header: `asio.hpp`

5.76.8 `basic_socket_streambuf::error`

Get the last error associated with the stream buffer.

```
const asio::error_code & error() const;
```

Return Value

An `error_code` corresponding to the last error from the stream buffer.

5.76.9 `basic_socket_streambuf::expires_after`

Set the stream buffer's expiry time relative to now.

```
void expires_after(
    const duration & 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.

5.76.10 `basic_socket_streambuf::expires_at`

(Deprecated: Use `expiry()`) Get the stream buffer's expiry time as an absolute time.

```
time_point expires_at() const;
```

Set the stream buffer's expiry time as an absolute time.

```
void expires_at(
    const time_point & expiry_time);
```

5.76.10.1 `basic_socket_streambuf::expires_at (1 of 2 overloads)`

(Deprecated: Use `expiry()`) Get the stream buffer's expiry time as an absolute time.

```
time_point expires_at() const;
```

Return Value

An absolute time value representing the stream buffer's expiry time.

5.76.10.2 `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_point & 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.

5.76.11 `basic_socket_streambuf::expires_from_now`

(Deprecated: Use `expiry()`) Get the stream buffer's expiry time relative to now.

```
duration expires_from_now() const;
```

(Deprecated: Use `expires_after()`) Set the stream buffer's expiry time relative to now.

```
void expires_from_now(
    const duration & expiry_time);
```

5.76.11.1 `basic_socket_streambuf::expires_from_now (1 of 2 overloads)`

(Deprecated: Use `expiry()`) Get the stream buffer's expiry time relative to now.

```
duration expires_from_now() const;
```

Return Value

A relative time value representing the stream buffer's expiry time.

5.76.11.2 `basic_socket_streambuf::expires_from_now (2 of 2 overloads)`

(Deprecated: Use `expires_after()`) Set the stream buffer's expiry time relative to now.

```
void expires_from_now(
    const duration & 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.

5.76.12 basic_socket_streambuf::expiry

Get the stream buffer's expiry time as an absolute time.

```
time_point expiry() const;
```

Return Value

An absolute time value representing the stream buffer's expiry time.

5.76.13 basic_socket_streambuf::operator=

Move-assign a **basic_socket_streambuf** from another.

```
basic_socket_streambuf & operator=(  
    basic_socket_streambuf && other);
```

5.76.14 basic_socket_streambuf::overflow

```
int_type overflow(  
    int_type c);
```

5.76.15 basic_socket_streambuf::protocol_type

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.76.16 basic_socket_streambuf::puberror

(Deprecated: Use `error()`.) 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.

5.76.17 basic_socket_streambuf::setbuf

```
std::streambuf * setbuf(  
    char_type * s,  
    std::streamsize n);
```

5.76.18 basic_socket_streambuf::socket

Get a reference to the underlying socket.

```
basic_socket< Protocol > & socket();
```

5.76.19 basic_socket_streambuf::sync

```
int sync();
```

5.76.20 basic_socket_streambuf::time_point

The time type.

```
typedef WaitTraits::time_point time_point;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.76.21 basic_socket_streambuf::time_type

(Deprecated: Use time_point.) The time type.

```
typedef WaitTraits::time_type time_type;
```

Requirements

Header: asio/basic_socket_streambuf.hpp

Convenience header: asio.hpp

5.76.22 basic_socket_streambuf::underflow

```
int_type underflow();
```

5.76.23 basic_socket_streambuf::~basic_socket_streambuf

Destructor flushes buffered data.

```
virtual ~basic_socket_streambuf();
```

5.77 basic_stream_socket

Provides stream-oriented socket functionality.

```
template<
    typename Protocol>
class basic_stream_socket :
    public basic_socket< Protocol >
```

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.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.

Name	Description
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_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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.

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_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.
release	Release ownership of the underlying native 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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
write_some	Write some data to the socket.
~basic_stream_socket	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

Name	Description
message_out_of_band	Process out-of-band data.
message_peek	Peek at incoming data without removing it from the input queue.

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`

5.77.1 basic_stream_socket::assign

Assign an existing native socket to the socket.

```
void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket);

void assign(
    const protocol_type & protocol,
    const native_handle_type & native_socket,
    asio::error_code & ec);
```

5.77.1.1 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);
```

5.77.1.2 basic_stream_socket::assign (2 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,
    asio::error_code & ec);
```

5.77.2 basic_stream_socket::async_connect

Inherited from `basic_socket`.

Start an asynchronous connect.

```
template<
    typename ConnectHandler>
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_context::post()`.

Example

```
void connect_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Connect succeeded.
    }
}

...

asio::ip::tcp::socket socket(io_context);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.async_connect(endpoint, connect_handler);
```

5.77.3 basic_stream_socket::async_read_some

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
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_context::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`.

5.77.4 basic_stream_socket::async_receive

Start an asynchronous receive.

```
template<  
    typename MutableBufferSequence,  
    typename ReadHandler>  
DEDUCED async_receive(  
    const MutableBufferSequence & buffers,  
    ReadHandler && handler);  
  
template<  
    typename MutableBufferSequence,  
    typename ReadHandler>  
DEDUCED async_receive(  
    const MutableBufferSequence & buffers,  
    socket_base::message_flags flags,  
    ReadHandler && handler);
```

5.77.4.1 basic_stream_socket::async_receive (1 of 2 overloads)

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
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_context::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`.

5.77.4.2 basic_stream_socket::async_receive (2 of 2 overloads)

Start an asynchronous receive.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
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_context::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`.

5.77.5 basic_stream_socket::async_send

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
DEDUCED async_send(
    const ConstBufferSequence & buffers,
    WriteHandler && handler);

template<
    typename ConstBufferSequence,
    typename WriteHandler>
DEDUCED async_send(
    const ConstBufferSequence & buffers,
    socket_base::message_flags flags,
    WriteHandler && handler);
```

5.77.5.1 basic_stream_socket::async_send (1 of 2 overloads)

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
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_context::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`.

5.77.5.2 basic_stream_socket::async_send (2 of 2 overloads)

Start an asynchronous send.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
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_context::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`.

5.77.6 basic_stream_socket::async_wait

Inherited from basic_socket.

Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
template<  
    typename WaitHandler>  
DEDUCED async_wait(  
    wait_type w,  
    WaitHandler && handler);
```

This function is used to perform an asynchronous wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

handler The handler to be called when the wait 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_context::post()`.

Example

```
void wait_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Wait succeeded.
    }
}

...

asio::ip::tcp::socket socket(io_context);
...
socket.async_wait(asio::ip::tcp::socket::wait_read, wait_handler);
```

5.77.7 basic_stream_socket::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
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_context::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`.

5.77.8 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;
```

5.77.8.1 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.

5.77.8.2 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.

5.77.9 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;
```

5.77.9.1 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.

5.77.9.2 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.

5.77.10 basic_stream_socket::basic_stream_socket

Construct a `basic_stream_socket` without opening it.

```
explicit basic_stream_socket(
    asio::io_context & io_context);
```

Construct and open a `basic_stream_socket`.

```
basic_stream_socket(
    asio::io_context & io_context,
    const protocol_type & protocol);
```

Construct a `basic_stream_socket`, opening it and binding it to the given local endpoint.

```
basic_stream_socket(
    asio::io_context & io_context,
    const endpoint_type & endpoint);
```

Construct a `basic_stream_socket` on an existing native socket.

```
basic_stream_socket(
    asio::io_context & io_context,
    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>
basic_stream_socket(
    basic_stream_socket< Protocol1 > && other,
    typename enable_if< is_convertible< Protocol1, Protocol >::value >::type * = 0);
```

5.77.10.1 `basic_stream_socket::basic_stream_socket (1 of 6 overloads)`

Construct a `basic_stream_socket` without opening it.

```
basic_stream_socket(
    asio::io_context & io_context);
```

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_context The `io_context` object that the stream socket will use to dispatch handlers for any asynchronous operations performed on the socket.

5.77.10.2 `basic_stream_socket::basic_stream_socket (2 of 6 overloads)`

Construct and open a `basic_stream_socket`.

```
basic_stream_socket(
    asio::io_context & io_context,
    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_context The `io_context` 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.

5.77.10.3 `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_context & io_context,
    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_context The `io_context` 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.

5.77.10.4 `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_context & io_context,
    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_context The `io_context` 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.

5.77.10.5 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_context&)` constructor.

5.77.10.6 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>
basic_stream_socket(
    basic_stream_socket< Protocol1 > && 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_context&)` constructor.

5.77.11 basic_stream_socket::bind

Bind the socket to the given local endpoint.

```
void bind(
    const endpoint_type & endpoint);

void bind(
    const endpoint_type & endpoint,
    asio::error_code & ec);
```

5.77.11.1 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_context);  
socket.open(asio::ip::tcp::v4());  
socket.bind(asio::ip::tcp::endpoint(  
    asio::ip::tcp::v4(), 12345));
```

5.77.11.2 basic_stream_socket::bind (2 of 2 overloads)

Inherited from `basic_socket`.

Bind the socket to the given local endpoint.

```
void 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_context);  
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.  
}
```

5.77.12 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_context);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.77.13 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_context);
...
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

5.77.14 basic_stream_socket::cancel

Cancel all asynchronous operations associated with the socket.

```
void cancel();  
  
void cancel(  
    asio::error_code & ec);
```

5.77.14.1 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.

5.77.14.2 basic_stream_socket::cancel (2 of 2 overloads)

Inherited from basic_socket.

Cancel all asynchronous operations associated with the socket.

```
void 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.

5.77.15 basic_stream_socket::close

Close the socket.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.77.15.1 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.

5.77.15.2 basic_stream_socket::close (2 of 2 overloads)

Inherited from `basic_socket`.

Close the socket.

```
void 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_context);  
...  
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.

5.77.16 basic_stream_socket::connect

Connect the socket to the specified endpoint.

```
void connect(  
    const endpoint_type & peer_endpoint);  
  
void connect(  
    const endpoint_type & peer_endpoint,  
    asio::error_code & ec);
```

5.77.16.1 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_context);
asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("1.2.3.4"), 12345);
socket.connect(endpoint);
```

5.77.16.2 basic_stream_socket::connect (2 of 2 overloads)

Inherited from basic_socket.

Connect the socket to the specified endpoint.

```
void 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_context);
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.
}
```

5.77.17 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_context);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.77.18 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_context);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.77.19 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_context);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.77.20 basic_stream_socket::endpoint_type

The endpoint type.

```
typedef Protocol::endpoint endpoint_type;
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.77.21 basic_stream_socket::executor_type

Inherited from basic_socket.

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.77.22 basic_stream_socket::get_executor

Inherited from basic_socket.

Get the executor associated with the object.

```
executor_type get_executor();
```

5.77.23 basic_stream_socket::get_io_context

Inherited from basic_socket.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.77.24 basic_stream_socket::get_io_service

Inherited from basic_socket.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.77.25 basic_stream_socket::get_option

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option) const;

template<
    typename GettableSocketOption>
void get_option(
    GettableSocketOption & option,
    asio::error_code & ec) const;
```

5.77.25.1 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_context);
...
asio::ip::tcp::socket::keep_alive option;
socket.get_option(option);
bool is_set = option.value();
```

5.77.25.2 basic_stream_socket::get_option (2 of 2 overloads)

Inherited from basic_socket.

Get an option from the socket.

```
template<
    typename GettableSocketOption>
void 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_context);
...
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();
```

5.77.26 basic_stream_socket::io_control

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.77.26.1 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_context);
...
asio::ip::tcp::socket::bytes_readable command;
socket.io_control(command);
std::size_t bytes_readable = command.get();
```

5.77.26.2 basic_stream_socket::io_control (2 of 2 overloads)

Inherited from basic_socket.

Perform an IO control command on the socket.

```
template<
    typename IoControlCommand>
void 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_context);
...
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();
```

5.77.27 basic_stream_socket::is_open

Inherited from basic_socket.

Determine whether the socket is open.

```
bool is_open() const;
```

5.77.28 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_context);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.77.29 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_context);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.77.30 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;
```

5.77.30.1 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_context);
...
asio::ip::tcp::endpoint endpoint = socket.local_endpoint();
```

5.77.30.2 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_context);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.local_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.77.31 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;
```

5.77.31.1 `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.

5.77.31.2 `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.

5.77.32 `basic_stream_socket::lowest_layer_type`

Inherited from basic_socket.

A `basic_socket` is always the lowest layer.

```
typedef basic_socket< Protocol > 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>executor_type</code>	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

Member Functions

Name	Description
assign	Assign an existing native socket to the socket.
async_connect	Start an asynchronous connect.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</code> associated with the object.
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</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_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>release</code>	Release ownership of the underlying native socket.

Name	Description
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

Protected Member Functions

Name	Description
<code>~basic_socket</code>	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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`

5.77.33 basic_stream_socket::max_connections

Inherited from socket_base.

(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.77.34 basic_stream_socket::max_listen_connections

Inherited from socket_base.

The maximum length of the queue of pending incoming connections.

```
static const int max_listen_connections = implementation_defined;
```

5.77.35 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;
```

5.77.36 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;
```

5.77.37 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

5.77.38 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;
```

5.77.39 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;
```

5.77.40 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.

5.77.41 basic_stream_socket::native_handle_type

The native representation of a socket.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.77.42 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);
```

```
void native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.77.42.1 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, op);
}

```

5.77.42.2 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, op);
}

```

5.77.42.3 basic_stream_socket::native_non_blocking (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the native socket implementation.

```
void 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_wait(tcp::socket::wait_write, *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_wait(tcp::socket::wait_write, op);
}

```

5.77.43 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);

void non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.77.43.1 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`.

5.77.43.2 `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`.

5.77.43.3 `basic_stream_socket::non_blocking` (3 of 3 overloads)

Inherited from basic_socket.

Sets the non-blocking mode of the socket.

```
void 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`.

5.77.44 basic_stream_socket::open

Open the socket using the specified protocol.

```
void open(
    const protocol_type & protocol = protocol_type());  
  
void open(
    const protocol_type & protocol,
    asio::error_code & ec);
```

5.77.44.1 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_context);
socket.open(asio::ip::tcp::v4());
```

5.77.44.2 basic_stream_socket::open (2 of 2 overloads)

Inherited from basic_socket.

Open the socket using the specified protocol.

```
void 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_context);
asio::error_code ec;
socket.open(asio::ip::tcp::v4(), ec);
if (ec)
{
    // An error occurred.
}
```

5.77.45 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>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_stream_socket >::type & operator <=
    (=(
        basic_stream_socket< Protocol1 > && other);
```

5.77.45.1 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_context&)` constructor.

5.77.45.2 basic_stream_socket::operator= (2 of 2 overloads)

Move-assign a `basic_stream_socket` from a socket of another protocol type.

```
template<
    typename Protocol1>
enable_if< is_convertible< Protocol1, Protocol >::value, basic_stream_socket >::type & operator <=
    (=(
        basic_stream_socket< Protocol1 > && 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_context&)` constructor.

5.77.46 `basic_stream_socket::out_of_band_inline`

Inherited from socket_base.

Socket option for putting received out-of-band data inline.

```
typedef implementation_defined out_of_band_inline;
```

Implements the SOL_SOCKET/SO_OOBINLINE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option;
socket.get_option(option);
bool value = option.value();
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.77.47 `basic_stream_socket::protocol_type`

The protocol type.

```
typedef Protocol protocol_type;
```

Requirements

Header: asio/basic_stream_socket.hpp

Convenience header: asio.hpp

5.77.48 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);
```

5.77.48.1 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`.

5.77.48.2 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.

5.77.49 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);
```

5.77.49.1 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`.

5.77.49.2 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`.

5.77.49.3 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.

5.77.50 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_context);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.77.51 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_context);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.77.52 basic_stream_socket::release

Release ownership of the underlying native socket.

```
native_handle_type release();

native_handle_type release(
    asio::error_code & ec);
```

5.77.52.1 basic_stream_socket::release (1 of 2 overloads)

Inherited from basic_socket.

Release ownership of the underlying native socket.

```
native_handle_type release();
```

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. Ownership of the native socket is then transferred to the caller.

Exceptions

asio::system_error Thrown on failure.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

5.77.52.2 basic_stream_socket::release (2 of 2 overloads)

Inherited from basic_socket.

Release ownership of the underlying native socket.

```
native_handle_type release(
    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. Ownership of the native socket is then transferred to the caller.

Parameters

ec Set to indicate what error occurred, if any.

Remarks

This function is unsupported on Windows versions prior to Windows 8.1, and will fail with `asio::error::operation_not_supported` on these platforms.

5.77.53 `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;
```

5.77.53.1 `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_context);  
...  
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint();
```

5.77.53.2 `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_context);
...
asio::error_code ec;
asio::ip::tcp::endpoint endpoint = socket.remote_endpoint(ec);
if (ec)
{
    // An error occurred.
}
```

5.77.54 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_context);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.77.55 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);
```

5.77.55.1 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.

5.77.55.2 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.

5.77.55.3 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.

5.77.56 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_context);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.77.57 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_context);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.77.58 basic_stream_socket::set_option

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option);

template<
    typename SettableSocketOption>
void set_option(
    const SettableSocketOption & option,
    asio::error_code & ec);
```

5.77.58.1 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_context);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

5.77.58.2 basic_stream_socket::set_option (2 of 2 overloads)

Inherited from basic_socket.

Set an option on the socket.

```
template<
    typename SettableSocketOption>
void 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_context);
...
asio::ip::tcp::no_delay option(true);
asio::error_code ec;
socket.set_option(option, ec);
if (ec)
{
    // An error occurred.
}
```

5.77.59 basic_stream_socket::shutdown

Disable sends or receives on the socket.

```
void shutdown(
    shutdown_type what);

void shutdown(
    shutdown_type what,
    asio::error_code & ec);
```

5.77.59.1 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_context);
...
socket.shutdown(asio::ip::tcp::socket::shutdown_send);
```

5.77.59.2 basic_stream_socket::shutdown (2 of 2 overloads)

Inherited from `basic_socket`.

Disable sends or receives on the socket.

```
void 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_context);
...
asio::error_code ec;
socket.shutdown(asio::ip::tcp::socket::shutdown_send, ec);
if (ec)
{
    // An error occurred.
}
```

5.77.60 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.

5.77.61 basic_stream_socket::wait

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(
    wait_type w);

void wait(
    wait_type w,
    asio::error_code & ec);
```

5.77.61.1 basic_stream_socket::wait (1 of 2 overloads)

Inherited from `basic_socket`.

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);
```

This function is used to perform a blocking wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

Example

Waiting for a socket to become readable.

```
asio::ip::tcp::socket socket(io_context);  
...  
socket.wait(asio::ip::tcp::socket::wait_read);
```

5.77.61.2 basic_stream_socket::wait (2 of 2 overloads)

Inherited from `basic_socket`.

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

This function is used to perform a blocking wait for a socket to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired socket state.

ec Set to indicate what error occurred, if any.

Example

Waiting for a socket to become readable.

```
asio::ip::tcp::socket socket(io_context);  
...  
asio::error_code ec;  
socket.wait(asio::ip::tcp::socket::wait_read, ec);
```

5.77.62 basic_stream_socket::wait_type

Inherited from `socket_base`.

Wait types.

```
enum wait_type
```

Values

wait_read Wait for a socket to become ready to read.

wait_write Wait for a socket to become ready to write.

wait_error Wait for a socket to have error conditions pending.

For use with `basic_socket::wait()` and `basic_socket::async_wait()`.

5.77.63 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);
```

5.77.63.1 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`.

5.77.63.2 `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.

5.77.64 `basic_stream_socket::~basic_stream_socket`

Destroys the socket.

```
~basic_stream_socket();
```

This function destroys the socket, cancelling any outstanding asynchronous operations associated with the socket as if by calling `cancel`.

5.78 `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
basic_streambuf	Construct a basic_streambuf object.
capacity	Get the current capacity of the basic_streambuf.
commit	Move characters from the output sequence to the input sequence.
consume	Remove characters from the input sequence.
data	Get a list of buffers that represents the input sequence.
max_size	Get the maximum size of the basic_streambuf.
prepare	Get a list of buffers that represents the output sequence, with the given size.
size	Get the size of the input sequence.

Protected Member Functions

Name	Description
overflow	Override std::streambuf behaviour.
reserve	
underflow	Override std::streambuf 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`

5.78.1 **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.

5.78.2 basic_streambuf::capacity

Get the current capacity of the `basic_streambuf`.

```
std::size_t capacity() const;
```

Return Value

The current total capacity of the streambuf, i.e. for both the input sequence and output sequence.

5.78.3 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 $x \geq n$, 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.

5.78.4 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`

5.78.5 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.

5.78.6 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.

5.78.7 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.

5.78.8 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`

5.78.9 basic_streambuf::overflow

Override `std::streambuf` behaviour.

```
int_type overflow(
    int_type c);
```

Behaves according to the specification of `std::basic_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.

5.78.10 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.

5.78.11 basic_streambuf::reserve

```
void reserve(  
    std::size_t n);
```

5.78.12 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 += buf.size();  
}
```

5.78.13 basic_streambuf::underflow

Override `std::streambuf` behaviour.

```
int_type underflow();
```

Behaves according to the specification of `std::streambuf::underflow()`.

5.79 basic_streambuf_ref

Adapts `basic_streambuf` to the dynamic buffer sequence type requirements.

```
template<  
    typename Allocator = std::allocator<char>>  
class basic_streambuf_ref
```

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
basic_streambuf_ref	Construct a basic_streambuf_ref for the given basic_streambuf object. Copy construct a basic_streambuf_ref. Move construct a basic_streambuf_ref.
capacity	Get the current capacity of the dynamic buffer.
commit	Move bytes from the output sequence to the input sequence.
consume	Remove characters from the input sequence.
data	Get a list of buffers that represents the input sequence.
max_size	Get the maximum size of the dynamic buffer.
prepare	Get a list of buffers that represents the output sequence, with the given size.
size	Get the size of the input sequence.

Requirements

Header: asio/basic_streambuf.hpp

Convenience header: asio.hpp

5.79.1 basic_streambuf_ref::basic_streambuf_ref

Construct a **basic_streambuf_ref** for the given **basic_streambuf** object.

```
explicit basic_streambuf_ref(
    basic_streambuf< Allocator > & sb);
```

Copy construct a **basic_streambuf_ref**.

```
basic_streambuf_ref(
    const basic_streambuf_ref & other);
```

Move construct a **basic_streambuf_ref**.

```
basic_streambuf_ref(
    basic_streambuf_ref && other);
```

5.79.1.1 `basic_streambuf_ref::basic_streambuf_ref (1 of 3 overloads)`

Construct a `basic_streambuf_ref` for the given `basic_streambuf` object.

```
basic_streambuf_ref(
    basic_streambuf< Allocator > & sb);
```

5.79.1.2 `basic_streambuf_ref::basic_streambuf_ref (2 of 3 overloads)`

Copy construct a `basic_streambuf_ref`.

```
basic_streambuf_ref(
    const basic_streambuf_ref & other);
```

5.79.1.3 `basic_streambuf_ref::basic_streambuf_ref (3 of 3 overloads)`

Move construct a `basic_streambuf_ref`.

```
basic_streambuf_ref(
    basic_streambuf_ref && other);
```

5.79.2 `basic_streambuf_ref::capacity`

Get the current capacity of the dynamic buffer.

```
std::size_t capacity() const;
```

5.79.3 `basic_streambuf_ref::commit`

Move bytes from the output sequence to the input sequence.

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

5.79.4 `basic_streambuf_ref::const_buffers_type`

The type used to represent the input sequence as a list of buffers.

```
typedef basic_streambuf< Allocator >::const_buffers_type const_buffers_type;
```

Types

Name	Description
<code>const_buffers_type</code>	The type used to represent the input sequence as a list of buffers.
<code>mutable_buffers_type</code>	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>capacity</code>	Get the current capacity of the <code>basic_streambuf</code> .
<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

5.79.5 basic_streambuf_ref::consume

Remove characters from the input sequence.

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

5.79.6 basic_streambuf_ref::data

Get a list of buffers that represents the input sequence.

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

5.79.7 basic_streambuf_ref::max_size

Get the maximum size of the dynamic buffer.

```
std::size_t max_size() const;
```

5.79.8 `basic_streambuf_ref::mutable_buffers_type`

The type used to represent the output sequence as a list of buffers.

```
typedef basic_streambuf< Allocator >::mutable_buffers_type mutable_buffers_type;
```

Types

Name	Description
<code>const_buffers_type</code>	The type used to represent the input sequence as a list of buffers.
<code>mutable_buffers_type</code>	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>capacity</code>	Get the current capacity of the <code>basic_streambuf</code> .
<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`

5.79.9 `basic_streambuf_ref::prepare`

Get a list of buffers that represents the output sequence, with the given size.

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

5.79.10 `basic_streambuf_ref::size`

Get the size of the input sequence.

```
std::size_t size() const;
```

5.80 `basic_waitable_timer`

Provides waitable timer functionality.

```
template<  
    typename Clock,  
    typename WaitTraits>  
class basic_waitable_timer
```

Types

Name	Description
<code>clock_type</code>	The clock type.
<code>duration</code>	The duration type of the clock.
<code>executor_type</code>	The type of the executor associated with the object.
<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_waitable_timer</code>	Constructor. Constructor to set a particular expiry time as an absolute time. Constructor to set a particular expiry time relative to now. Move-construct a <code>basic_waitable_timer</code> from another.
<code>cancel</code>	Cancel any asynchronous operations that are waiting on the timer. (Deprecated: Use <code>non-error_code</code> overload.) Cancel any asynchronous operations that are waiting on the timer.

Name	Description
<code>cancel_one</code>	Cancels one asynchronous operation that is waiting on the timer. (Deprecated: Use non-error_code overload.) Cancels one asynchronous operation that is waiting on the timer.
<code>expires_after</code>	Set the timer's expiry time relative to now.
<code>expires_at</code>	(Deprecated: Use expiry().) Get the timer's expiry time as an absolute time. Set the timer's expiry time as an absolute time. (Deprecated: Use non-error_code overload.) Set the timer's expiry time as an absolute time.
<code>expires_from_now</code>	(Deprecated: Use expiry().) Get the timer's expiry time relative to now. (Deprecated: Use expires_after().) Set the timer's expiry time relative to now.
<code>expiry</code>	Get the timer's expiry time as an absolute time.
<code>get_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use get_executor().) Get the io_context associated with the object.
<code>get_io_service</code>	(Deprecated: Use get_executor().) Get the io_context associated with the object.
<code>operator=</code>	Move-assign a basic_waitable_timer from another.
<code>wait</code>	Perform a blocking wait on the timer.
<code>~basic_waitable_timer</code>	Destroys the timer.

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_context);  
  
// Set an expiry time relative to now.  
timer.expires_after(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_context,  
    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_after(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_after()` 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

5.80.1 basic_waitable_timer::async_wait

Start an asynchronous wait on the timer.

```
template<
    typename WaitHandler>
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_context::post()`.

5.80.2 basic_waitable_timer::basic_waitable_timer

Constructor.

```
explicit basic_waitable_timer(
    asio::io_context & io_context);
```

Constructor to set a particular expiry time as an absolute time.

```
basic_waitable_timer(
    asio::io_context & io_context,
    const time_point & expiry_time);
```

Constructor to set a particular expiry time relative to now.

```
basic_waitable_timer(
    asio::io_context & io_context,
    const duration & expiry_time);
```

Move-construct a `basic_waitable_timer` from another.

```
basic_waitable_timer(
    basic_waitable_timer && other);
```

5.80.2.1 `basic_waitable_timer::basic_waitable_timer (1 of 4 overloads)`

Constructor.

```
basic_waitable_timer(
    asio::io_context & io_context);
```

This constructor creates a timer without setting an expiry time. The `expires_at()` or `expires_after()` functions must be called to set an expiry time before the timer can be waited on.

Parameters

io_context The `io_context` object that the timer will use to dispatch handlers for any asynchronous operations performed on the timer.

5.80.2.2 `basic_waitable_timer::basic_waitable_timer (2 of 4 overloads)`

Constructor to set a particular expiry time as an absolute time.

```
basic_waitable_timer(
    asio::io_context & io_context,
    const time_point & expiry_time);
```

This constructor creates a timer and sets the expiry time.

Parameters

io_context The `io_context` 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.

5.80.2.3 `basic_waitable_timer::basic_waitable_timer (3 of 4 overloads)`

Constructor to set a particular expiry time relative to now.

```
basic_waitable_timer(
    asio::io_context & io_context,
    const duration & expiry_time);
```

This constructor creates a timer and sets the expiry time.

Parameters

io_context The `io_context` 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.

5.80.2.4 `basic_waitable_timer::basic_waitable_timer` (4 of 4 overloads)

Move-construct a `basic_waitable_timer` from another.

```
basic_waitable_timer(
    basic_waitable_timer && other);
```

This constructor moves a timer from one object to another.

Parameters

other The other `basic_waitable_timer` 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_waitable_timer(io_context&)` constructor.

5.80.3 `basic_waitable_timer::cancel`

Cancel any asynchronous operations that are waiting on the timer.

```
std::size_t cancel();
```

(Deprecated: Use non-error_code overload.) Cancel any asynchronous operations that are waiting on the timer.

```
std::size_t cancel(
    asio::error_code & ec);
```

5.80.3.1 `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.

5.80.3.2 `basic_waitable_timer::cancel (2 of 2 overloads)`

(Deprecated: Use non-error_code overload.) 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.

5.80.4 `basic_waitable_timer::cancel_one`

Cancels one asynchronous operation that is waiting on the timer.

```
std::size_t cancel_one();
```

(Deprecated: Use non-error_code overload.) Cancels one asynchronous operation that is waiting on the timer.

```
std::size_t cancel_one(  
    asio::error_code & ec);
```

5.80.4.1 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.

5.80.4.2 basic_waitable_timer::cancel_one (2 of 2 overloads)

(Deprecated: Use non-error_code overload.) 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.

5.80.5 `basic_waitable_timer::clock_type`

The clock type.

```
typedef Clock clock_type;
```

Requirements

Header: `asio/basic_waitable_timer.hpp`

Convenience header: `asio.hpp`

5.80.6 `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`

5.80.7 `basic_waitable_timer::executor_type`

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
<code>context</code>	Obtain the underlying execution context.
<code>defer</code>	Request the <code>io_context</code> to invoke the given function object.
<code>dispatch</code>	Request the <code>io_context</code> to invoke the given function object.

Name	Description
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/basic_waitable_timer.hpp

Convenience header: asio.hpp

5.80.8 basic_waitable_timer::expires_after

Set the timer's expiry time relative to now.

```
std::size_t expires_after(
    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_after()` 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.

5.80.9 `basic_waitable_timer::expires_at`

(Deprecated: Use `expiry()`.) 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);
```

(Deprecated: Use non-error_code overload.) Set the timer's expiry time as an absolute time.

```
std::size_t expires_at(
    const time_point & expiry_time,
    asio::error_code & ec);
```

5.80.9.1 `basic_waitable_timer::expires_at (1 of 3 overloads)`

(Deprecated: Use `expiry()`.) 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.

5.80.9.2 `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.

5.80.9.3 `basic_waitable_timer::expires_at (3 of 3 overloads)`

(Deprecated: Use non-error_code overload.) 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.

5.80.10 `basic_waitable_timer::expires_from_now`

(Deprecated: Use `expiry()`.) Get the timer's expiry time relative to now.

```
duration expires_from_now() const;
```

(Deprecated: Use `expires_after()`.) 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);
```

5.80.10.1 `basic_waitable_timer::expires_from_now (1 of 3 overloads)`

(Deprecated: Use `expiry()`.) 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.

5.80.10.2 `basic_waitable_timer::expires_from_now (2 of 3 overloads)`

(Deprecated: Use `expires_after()`.) 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.

5.80.10.3 `basic_waitable_timer::expires_from_now` (3 of 3 overloads)

(Deprecated: Use `expires_after()`.) 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.

5.80.11 `basic_waitable_timer::expiry`

Get the timer's expiry time as an absolute time.

```
time_point expiry() 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.

5.80.12 `basic_waitable_timer::get_executor`

Get the executor associated with the object.

```
executor_type get_executor();
```

5.80.13 `basic_waitable_timer::get_io_context`

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.80.14 basic_waitable_timer::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.80.15 basic_waitable_timer::operator=

Move-assign a `basic_waitable_timer` from another.

```
basic_waitable_timer & operator=(  
    basic_waitable_timer && other);
```

This assignment operator moves a timer from one object to another. Cancels any outstanding asynchronous operations associated with the target object.

Parameters

other The other `basic_waitable_timer` 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_waitable_timer(io_context&)` constructor.

5.80.16 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`

5.80.17 basic_waitable_timer::traits_type

The wait traits type.

```
typedef WaitTraits traits_type;
```

Requirements

Header: asio/basic_waitable_timer.hpp

Convenience header: asio.hpp

5.80.18 basic_waitable_timer::wait

Perform a blocking wait on the timer.

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

5.80.18.1 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.

5.80.18.2 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.

5.80.19 basic_waitable_timer::~basic_waitable_timer

Destroys the timer.

```
~basic_waitable_timer();
```

This function destroys the timer, cancelling any outstanding asynchronous wait operations associated with the timer as if by calling `cancel`.

5.81 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
basic_yield_context	Construct a yield context to represent the specified coroutine. Construct a yield context from another yield context type.
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

5.81.1 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);
```

Construct a yield context from another yield context type.

```
template<
    typename OtherHandler>
basic_yield_context(
    const basic_yield_context< OtherHandler > & other);
```

5.81.1.1 `basic_yield_context::basic_yield_context (1 of 2 overloads)`

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.

5.81.1.2 `basic_yield_context::basic_yield_context (2 of 2 overloads)`

Construct a yield context from another yield context type.

```
template<
    typename OtherHandler>
basic_yield_context(
    const basic_yield_context< OtherHandler > & other);
```

Requires that `OtherHandler` be convertible to `Handler`.

5.81.2 `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

5.81.3 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

5.81.4 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.
    }
    ...
}
```

5.82 bind_executor

Associate an object of type `T` with an executor of type `Executor`.

```
template<
    typename Executor,
    typename T>
executor_binder< typename decay< T >::type, Executor > bind_executor(
    const Executor & ex,
    T && t,
    typename enable_if< is_executor< Executor >::value >::type * = 0);

template<
    typename ExecutionContext,
```

```

typename T>
executor_binder< typename decay< T >::type, typename ExecutionContext::executor_type > ←
bind_executor(
    ExecutionContext & ctx,
    T && t,
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >:: ←
        type * = 0);

```

Requirements

Header: asio/bind_executor.hpp

Convenience header: asio.hpp

5.82.1 bind_executor (1 of 2 overloads)

Associate an object of type `T` with an executor of type `Executor`.

```

template<
    typename Executor,
    typename T>
executor_binder< typename decay< T >::type, Executor > bind_executor(
    const Executor & ex,
    T && t,
    typename enable_if< is_executor< Executor >::value >::type * = 0);

```

5.82.2 bind_executor (2 of 2 overloads)

Associate an object of type `T` with an execution context's executor.

```

template<
    typename ExecutionContext,
    typename T>
executor_binder< typename decay< T >::type, typename ExecutionContext::executor_type > ←
bind_executor(
    ExecutionContext & ctx,
    T && t,
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >:: ←
        type * = 0);

```

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_buffer buffer(
    const mutable_buffer & b);

mutable_buffer buffer(
    const mutable_buffer & b,
    std::size_t max_size_in_bytes);

const_buffer buffer(

```

```

const const_buffer & b);

const_buffer buffer(
    const const_buffer & b,
    std::size_t max_size_in_bytes);

mutable_buffer buffer(
    void * data,
    std::size_t size_in_bytes);

const_buffer buffer(
    const void * data,
    std::size_t size_in_bytes);

template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    PodType (&data) [N]);

template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    PodType (&data) [N],
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const PodType (&data) [N]);

template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const PodType (&data) [N],
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    boost::array< PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    boost::array< PodType, N > & data,
    std::size_t max_size_in_bytes);

```

```

template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    boost::array< const PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    boost::array< const PodType, N > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const boost::array< PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const boost::array< PodType, N > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    std::array< PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    std::array< PodType, N > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    std::array< const PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    std::array< const PodType, N > & data,
    std::size_t max_size_in_bytes);

template<

```

```

typename PodType,
std::size_t N>
const_buffer buffer(
    const std::array< PodType, N > & data);

template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const std::array< PodType, N > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    typename Allocator>
mutable_buffer buffer(
    std::vector< PodType, Allocator > & data);

template<
    typename PodType,
    typename Allocator>
mutable_buffer buffer(
    std::vector< PodType, Allocator > & data,
    std::size_t max_size_in_bytes);

template<
    typename PodType,
    typename Allocator>
const_buffer buffer(
    const std::vector< PodType, Allocator > & data);

template<
    typename PodType,
    typename Allocator>
const_buffer buffer(
    const std::vector< PodType, Allocator > & data,
    std::size_t max_size_in_bytes);

template<
    typename Elem,
    typename Traits,
    typename Allocator>
mutable_buffer buffer(
    std::basic_string< Elem, Traits, Allocator > & data);

template<
    typename Elem,
    typename Traits,
    typename Allocator>
mutable_buffer buffer(
    std::basic_string< Elem, Traits, Allocator > & data,
    std::size_t max_size_in_bytes);

template<

```

```

typename Elem,
typename Traits,
typename Allocator>
const_buffer buffer(
    const std::basic_string< Elem, Traits, Allocator > & data);

template<
    typename Elem,
    typename Traits,
    typename Allocator>
const_buffer buffer(
    const std::basic_string< Elem, Traits, Allocator > & data,
    std::size_t max_size_in_bytes);

template<
    typename Elem,
    typename Traits>
const_buffer buffer(
    basic_string_view< Elem, Traits > data);

template<
    typename Elem,
    typename Traits>
const_buffer buffer(
    basic_string_view< Elem, Traits > 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 `data()` and `size()` member functions:

```
asio::mutable_buffer b1 = ...;
std::size_t s1 = b1.size();
unsigned char* p1 = static_cast<unsigned char*>(b1.data());
```



```
asio::const_buffer b2 = ...;
std::size_t s2 = b2.size();
const void* p2 = b2.data();
```

The `data()` member function permits violations of type safety, so uses of it in application code should be carefully considered.

For convenience, a `buffer_size` function is provided that works with both buffers and 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` function, 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 32 overloads)

Create a new modifiable buffer from an existing buffer.

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

Return Value

```
mutable_buffer(b).
```

5.83.2 buffer (2 of 32 overloads)

Create a new modifiable buffer from an existing buffer.

```
mutable_buffer buffer(
    const mutable_buffer & b,
    std::size_t max_size_in_bytes);
```

Return Value

A [mutable_buffer](#) value equivalent to:

```
mutable_buffer(
    b.data(),
    min(b.size(), max_size_in_bytes));
```

5.83.3 buffer (3 of 32 overloads)

Create a new non-modifiable buffer from an existing buffer.

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

Return Value

`const_buffer(b).`

5.83.4 buffer (4 of 32 overloads)

Create a new non-modifiable buffer from an existing buffer.

```
const_buffer buffer(
    const const_buffer & b,
    std::size_t max_size_in_bytes);
```

Return Value

A [const_buffer](#) value equivalent to:

```
const_buffer(
    b.data(),
    min(b.size(), max_size_in_bytes));
```

5.83.5 buffer (5 of 32 overloads)

Create a new modifiable buffer that represents the given memory range.

```
mutable_buffer buffer(
    void * data,
    std::size_t size_in_bytes);
```

Return Value

```
mutable_buffer(data, size_in_bytes).
```

5.83.6 buffer (6 of 32 overloads)

Create a new non-modifiable buffer that represents the given memory range.

```
const_buffer buffer(
    const void * data,
    std::size_t size_in_bytes);
```

Return Value

```
const_buffer(data, size_in_bytes).
```

5.83.7 buffer (7 of 32 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    PodType (&data) [N]);
```

Return Value

A [mutable_buffer](#) value equivalent to:

```
mutable_buffer(
    static_cast<void*>(data),
    N * sizeof(PodType));
```

5.83.8 buffer (8 of 32 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    PodType (&data) [N],
    std::size_t max_size_in_bytes);
```

Return Value

A [mutable_buffer](#) value equivalent to:

```
mutable_buffer(
    static_cast<void*>(data),
    min(N * sizeof(PodType), max_size_in_bytes));
```

5.83.9 buffer (9 of 32 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const PodType (&data) [N]);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    static_cast<const void*>(data),
    N * sizeof(PodType));
```

5.83.10 buffer (10 of 32 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const PodType (&data) [N],
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    static_cast<const void*>(data),
    min(N * sizeof(PodType), max_size_in_bytes));
```

5.83.11 buffer (11 of 32 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    boost::array< PodType, N > & data);
```

Return Value

A `mutable_buffer` value equivalent to:

```
mutable_buffer(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.12 buffer (12 of 32 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    boost::array< PodType, N > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `mutable_buffer` value equivalent to:

```
mutable_buffer(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.13 buffer (13 of 32 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    boost::array< const PodType, N > & data);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.14 buffer (14 of 32 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    boost::array< const PodType, N > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.15 buffer (15 of 32 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const boost::array< PodType, N > & data);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.16 buffer (16 of 32 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const boost::array< PodType, N > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.17 buffer (17 of 32 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    std::array< PodType, N > & data);
```

Return Value

A `mutable_buffer` value equivalent to:

```
mutable_buffer(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.18 buffer (18 of 32 overloads)

Create a new modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
mutable_buffer buffer(
    std::array< PodType, N > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `mutable_buffer` value equivalent to:

```
mutable_buffer(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.19 buffer (19 of 32 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    std::array< const PodType, N > & data);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.20 buffer (20 of 32 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    std::array< const PodType, N > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.21 buffer (21 of 32 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const std::array<PodType, N> & data);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    data.data(),
    data.size() * sizeof(PodType));
```

5.83.22 buffer (22 of 32 overloads)

Create a new non-modifiable buffer that represents the given POD array.

```
template<
    typename PodType,
    std::size_t N>
const_buffer buffer(
    const std::array<PodType, N> & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    data.data(),
    min(data.size() * sizeof(PodType), max_size_in_bytes));
```

5.83.23 buffer (23 of 32 overloads)

Create a new modifiable buffer that represents the given POD vector.

```
template<
    typename PodType,
    typename Allocator>
mutable_buffer buffer(
    std::vector<PodType, Allocator> & data);
```

Return Value

A `mutable_buffer` value equivalent to:

```
mutable_buffer(
    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 32 overloads)**

Create a new modifiable buffer that represents the given POD vector.

```
template<
    typename PodType,
    typename Allocator>
mutable_buffer buffer(
    std::vector< PodType, Allocator > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A **mutable_buffer** value equivalent to:

```
mutable_buffer(
    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 32 overloads)**

Create a new non-modifiable buffer that represents the given POD vector.

```
template<
    typename PodType,
    typename Allocator>
const_buffer buffer(
    const std::vector< PodType, Allocator > & data);
```

Return Value

A **const_buffer** value equivalent to:

```
const_buffer(
    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 32 overloads)

Create a new non-modifiable buffer that represents the given POD vector.

```
template<
    typename PodType,
    typename Allocator>
const_buffer buffer(
    const std::vector< PodType, Allocator > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    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 32 overloads)

Create a new modifiable buffer that represents the given string.

```
template<
    typename Elem,
    typename Traits,
    typename Allocator>
mutable_buffer buffer(
    std::basic_string< Elem, Traits, Allocator > & data);
```

Return Value

```
mutable_buffer(data.size() ? &data[0] : 0, 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 32 overloads)

Create a new non-modifiable buffer that represents the given string.

```
template<
    typename Elem,
    typename Traits,
    typename Allocator>
mutable_buffer buffer(
    std::basic_string< Elem, Traits, Allocator > & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `mutable_buffer` value equivalent to:

```
mutable_buffer(
    data.size() ? &data[0] : 0,
    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.

5.83.29 `buffer` (29 of 32 overloads)

Create a new non-modifiable buffer that represents the given string.

```
template<
    typename Elem,
    typename Traits,
    typename Allocator>
const_buffer buffer(
    const std::basic_string<Elem, Traits, Allocator> & data);
```

Return Value

```
const_buffer(data.data(), data.size() * sizeof(Elem)).
```

Remarks

The buffer is invalidated by any non-const operation called on the given string object.

5.83.30 `buffer` (30 of 32 overloads)

Create a new non-modifiable buffer that represents the given string.

```
template<
    typename Elem,
    typename Traits,
    typename Allocator>
const_buffer buffer(
    const std::basic_string<Elem, Traits, Allocator> & data,
    std::size_t max_size_in_bytes);
```

Return Value

A `const_buffer` value equivalent to:

```
const_buffer(
    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.

5.83.31 buffer (31 of 32 overloads)

Create a new modifiable buffer that represents the given `string_view`.

```
template<
    typename Elem,
    typename Traits>
const_buffer buffer(
    basic_string_view< Elem, Traits > data);
```

Return Value

```
mutable_buffer(data.size() ? &data[0] : 0, data.size() * sizeof(Elem)).
```

5.83.32 buffer (32 of 32 overloads)

Create a new non-modifiable buffer that represents the given string.

```
template<
    typename Elem,
    typename Traits>
const_buffer buffer(
    basic_string_view< Elem, Traits > data,
    std::size_t max_size_in_bytes);
```

Return Value

A `mutable_buffer` value equivalent to:

```
mutable_buffer(
    data.size() ? &data[0] : 0,
    min(data.size() * sizeof(Elem), max_size_in_bytes));
```

5.84 buffer_cast

(Deprecated: Use the `data()` member function.) 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).

```
template<
    typename MutableBufferSequence,
    typename ConstBufferSequence>
std::size_t buffer_copy(
    const MutableBufferSequence & target,
    const ConstBufferSequence & source);

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 2 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.

5.85.2 buffer_copy (2 of 2 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_sequence_begin

The `asio::buffer_sequence_begin` function returns an iterator pointing to the first element in a buffer sequence.

```
const mutable_buffer * buffer_sequence_begin(
    const mutable_buffer & b);

const const_buffer * buffer_sequence_begin(
    const const_buffer & b);

template<
    typename C>
auto buffer_sequence_begin(
    C & c);

template<
    typename C>
auto buffer_sequence_begin(
    const C & c);
```

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.86.1 buffer_sequence_begin (1 of 4 overloads)

Get an iterator to the first element in a buffer sequence.

```
const mutable_buffer * buffer_sequence_begin(
    const mutable_buffer & b);
```

5.86.2 buffer_sequence_begin (2 of 4 overloads)

Get an iterator to the first element in a buffer sequence.

```
const const_buffer * buffer_sequence_begin(
    const const_buffer & b);
```

5.86.3 buffer_sequence_begin (3 of 4 overloads)

Get an iterator to the first element in a buffer sequence.

```
template<
    typename C>
auto buffer_sequence_begin(
    C & c);
```

5.86.4 buffer_sequence_begin (4 of 4 overloads)

Get an iterator to the first element in a buffer sequence.

```
template<
    typename C>
auto buffer_sequence_begin(
    const C & c);
```

5.87 buffer_sequence_end

The asio::buffer_sequence_end function returns an iterator pointing to one past the end element in a buffer sequence.

```
const mutable_buffer * buffer_sequence_end(
    const mutable_buffer & b);
```

```
const const_buffer * buffer_sequence_end(
    const const_buffer & b);
```

```
template<
    typename C>
auto buffer_sequence_end(
    C & c);
```

```
template<
    typename C>
auto buffer_sequence_end(
    const C & c);
```

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.87.1 `buffer_sequence_end` (1 of 4 overloads)

Get an iterator to one past the end element in a buffer sequence.

```
const mutable_buffer * buffer_sequence_end(
    const mutable_buffer & b);
```

5.87.2 `buffer_sequence_end` (2 of 4 overloads)

Get an iterator to one past the end element in a buffer sequence.

```
const const_buffer * buffer_sequence_end(
    const const_buffer & b);
```

5.87.3 `buffer_sequence_end` (3 of 4 overloads)

Get an iterator to one past the end element in a buffer sequence.

```
template<
    typename C>
auto buffer_sequence_end(
    C & c);
```

5.87.4 `buffer_sequence_end` (4 of 4 overloads)

Get an iterator to one past the end element in a buffer sequence.

```
template<
    typename C>
auto buffer_sequence_end(
    const C & c);
```

5.88 `buffer_size`

Get the total number of bytes in a buffer sequence.

```
template<
    typename BufferSequence>
std::size_t buffer_size(
    const BufferSequence & b);
```

The `buffer_size` function determines the total size of all buffers in the buffer sequence, as if computed as follows:

```
size_t total_size = 0;
auto i = asio::buffer_sequence_begin(buffers);
auto end = asio::buffer_sequence_end(buffers);
for (; i != end; ++i)
{
    const_buffer b(*i);
    total_size += b.size();
}
return total_size;
```

The `BufferSequence` template parameter may meet either of the `ConstBufferSequence` or `MutableBufferSequence` type requirements.

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.89 buffered_read_stream

Adds buffering to the read-related operations of a stream.

```
template<
    typename Stream>
class buffered_read_stream :
    noncopyable
```

Types

Name	Description
<code>executor_type</code>	The type of the executor associated with the object.
<code>lowest_layer_type</code>	The type of the lowest layer.
<code>next_layer_type</code>	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.

Name	Description
buffered_read_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.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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.

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

5.89.1 buffered_read_stream::async_fill

Start an asynchronous fill.

```
template<
    typename ReadHandler>
DEDUCED async_fill(
    ReadHandler && handler);
```

5.89.2 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>
DEDUCED async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler && handler);
```

5.89.3 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>
DEDUCED async_write_some(
    const ConstBufferSequence & buffers,
    WriteHandler && handler);
```

5.89.4 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);
```

5.89.4.1 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);
```

5.89.4.2 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);
```

5.89.5 buffered_read_stream::close

Close the stream.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.89.5.1 buffered_read_stream::close (1 of 2 overloads)

Close the stream.

```
void close();
```

5.89.5.2 buffered_read_stream::close (2 of 2 overloads)

Close the stream.

```
void close(  
    asio::error_code & ec);
```

5.89.6 buffered_read_stream::default_buffer_size

The default buffer size.

```
static const std::size_t default_buffer_size = implementation_defined;
```

5.89.7 buffered_read_stream::executor_type

The type of the executor associated with the object.

```
typedef lowest_layer_type::executor_type executor_type;
```

Requirements

Header: asio/buffered_read_stream.hpp

Convenience header: asio.hpp

5.89.8 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);
```

5.89.8.1 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();
```

5.89.8.2 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);
```

5.89.9 buffered_read_stream::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.89.10 buffered_read_stream::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

5.89.11 buffered_read_stream::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

5.89.12 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);
```

5.89.12.1 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();
```

5.89.12.2 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);
```

5.89.13 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;
```

5.89.13.1 buffered_read_stream::lowest_layer (1 of 2 overloads)

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

5.89.13.2 buffered_read_stream::lowest_layer (2 of 2 overloads)

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.89.14 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

5.89.15 buffered_read_stream::next_layer

Get a reference to the next layer.

```
next_layer_type & next_layer();
```

5.89.16 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

5.89.17 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);
```

5.89.17.1 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);
```

5.89.17.2 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);
```

5.89.18 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);
```

5.89.18.1 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);
```

5.89.18.2 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);
```

5.89.19 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);
```

5.89.19.1 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);
```

5.89.19.2 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);
```

5.90 buffered_stream

Adds buffering to the read- and write-related operations of a stream.

```
template<
    typename Stream>
class buffered_stream :
    noncopyable
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
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.

Name	Description
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.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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.

Name	Description
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.

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`

5.90.1 buffered_stream::async_fill

Start an asynchronous fill.

```
template<
    typename ReadHandler>
DEDUCED async_fill(
    ReadHandler && handler);
```

5.90.2 buffered_stream::async_flush

Start an asynchronous flush.

```
template<
    typename WriteHandler>
DEDUCED async_flush(
    WriteHandler && handler);
```

5.90.3 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>
DEDUCED async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler && handler);
```

5.90.4 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>
DEDUCED async_write_some(
    const ConstBufferSequence & buffers,
    WriteHandler && handler);
```

5.90.5 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);
```

5.90.5.1 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);
```

5.90.5.2 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);
```

5.90.6 buffered_stream::close

Close the stream.

```
void close();

void close(
    asio::error_code & ec);
```

5.90.6.1 buffered_stream::close (1 of 2 overloads)

Close the stream.

```
void close();
```

5.90.6.2 buffered_stream::close (2 of 2 overloads)

Close the stream.

```
void close(  
   asio::error_code & ec);
```

5.90.7 buffered_stream::executor_type

The type of the executor associated with the object.

```
typedef lowest_layer_type::executor_type executor_type;
```

Requirements

Header: asio/buffered_stream.hpp

Convenience header: asio.hpp

5.90.8 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);
```

5.90.8.1 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();
```

5.90.8.2 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);
```

5.90.9 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);
```

5.90.9.1 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();
```

5.90.9.2 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);
```

5.90.10 buffered_stream::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.90.11 buffered_stream::get_io_context

(Deprecated: Use `get_executor()`.) Get the **io_context** associated with the object.

```
asio::io_context & get_io_context();
```

5.90.12 buffered_stream::get_io_service

(Deprecated: Use `get_executor()`.) Get the **io_context** associated with the object.

```
asio::io_context & get_io_service();
```

5.90.13 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);
```

5.90.13.1 buffered_stream::in_avail (1 of 2 overloads)

Determine the amount of data that may be read without blocking.

```
std::size_t in_avail();
```

5.90.13.2 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);
```

5.90.14 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;
```

5.90.14.1 buffered_stream::lowest_layer (1 of 2 overloads)

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

5.90.14.2 buffered_stream::lowest_layer (2 of 2 overloads)

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.90.15 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

5.90.16 buffered_stream::next_layer

Get a reference to the next layer.

```
next_layer_type & next_layer();
```

5.90.17 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

5.90.18 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);
```

5.90.18.1 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);
```

5.90.18.2 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);
```

5.90.19 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);
```

5.90.19.1 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);
```

5.90.19.2 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);
```

5.90.20 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);
```

5.90.20.1 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);
```

5.90.20.2 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);
```

5.91 buffered_write_stream

Adds buffering to the write-related operations of a stream.

```
template<
    typename Stream>
class buffered_write_stream :
    noncopyable
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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`

5.91.1 buffered_write_stream::async_flush

Start an asynchronous flush.

```
template<
    typename WriteHandler>
DEDUCED async_flush(
    WriteHandler && handler);
```

5.91.2 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>
DEDUCED async_read_some(
    const MutableBufferSequence & buffers,
    ReadHandler && handler);
```

5.91.3 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>
DEDUCED async_write_some(
    const ConstBufferSequence & buffers,
    WriteHandler && handler);
```

5.91.4 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);
```

5.91.4.1 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);
```

5.91.4.2 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);
```

5.91.5 buffered_write_stream::close

Close the stream.

```
void close();
```



```
void close(
    asio::error_code & ec);
```

5.91.5.1 buffered_write_stream::close (1 of 2 overloads)

Close the stream.

```
void close();
```

5.91.5.2 buffered_write_stream::close (2 of 2 overloads)

Close the stream.

```
void close(
    asio::error_code & ec);
```

5.91.6 buffered_write_stream::default_buffer_size

The default buffer size.

```
static const std::size_t default_buffer_size = implementation_defined;
```

5.91.7 buffered_write_stream::executor_type

The type of the executor associated with the object.

```
typedef lowest_layer_type::executor_type executor_type;
```

Requirements

Header: asio/buffered_write_stream.hpp

Convenience header: asio.hpp

5.91.8 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);
```

5.91.8.1 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();
```

5.91.8.2 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);
```

5.91.9 buffered_write_stream::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.91.10 buffered_write_stream::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

5.91.11 buffered_write_stream::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

5.91.12 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);
```

5.91.12.1 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();
```

5.91.12.2 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);
```

5.91.13 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;
```

5.91.13.1 buffered_write_stream::lowest_layer (1 of 2 overloads)

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();
```

5.91.13.2 buffered_write_stream::lowest_layer (2 of 2 overloads)

Get a const reference to the lowest layer.

```
const lowest_layer_type & lowest_layer() const;
```

5.91.14 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

5.91.15 buffered_write_stream::next_layer

Get a reference to the next layer.

```
next_layer_type & next_layer();
```

5.91.16 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

5.91.17 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);
```

5.91.17.1 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);
```

5.91.17.2 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);
```

5.91.18 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);
```

5.91.18.1 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);
```

5.91.18.2 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);
```

5.91.19 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);
```

5.91.19.1 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);
```

5.91.19.2 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);
```

5.92 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

5.93 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

5.94 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.

Name	Description
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.
operator==	Test two iterators for equality.
operator>	Compare two iterators.
operator>=	Compare two iterators.

Requirements

Header: asio/buffers_iterator.hpp

Convenience header: asio.hpp

5.94.1 buffers_iterator::begin

Construct an iterator representing the beginning of the buffers' data.

```
static buffers_iterator begin(
    const BufferSequence & buffers);
```

5.94.2 buffers_iterator::buffers_iterator

Default constructor. Creates an iterator in an undefined state.

```
buffers_iterator();
```

5.94.3 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

5.94.4 buffers_iterator::end

Construct an iterator representing the end of the buffers' data.

```
static buffers_iterator end(
    const BufferSequence & buffers);
```

5.94.5 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

5.94.6 buffers_iterator::operator *

Dereference an iterator.

```
reference operator *() const;
```

5.94.7 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

5.94.8 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);
```

5.94.8.1 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

5.94.8.2 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

5.94.9 buffers_iterator::operator++

Increment operator (prefix).

```
buffers_iterator & operator++();
```

Increment operator (postfix).

```
buffers_iterator operator++(
    int );
```

5.94.9.1 buffers_iterator::operator++ (1 of 2 overloads)

Increment operator (prefix).

```
buffers_iterator & operator++();
```

5.94.9.2 buffers_iterator::operator++ (2 of 2 overloads)

Increment operator (postfix).

```
buffers_iterator operator++(int);
```

5.94.10 buffers_iterator::operator+=

Addition operator.

```
buffers_iterator & operator+=(std::ptrdiff_t difference);
```

5.94.11 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);
```

5.94.11.1 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

5.94.11.2 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

5.94.12 buffers_iterator::operator--

Decrement operator (prefix).

```
buffers_iterator & operator--();
```

Decrement operator (postfix).

```
buffers_iterator operator--(int);
```

5.94.12.1 buffers_iterator::operator-- (1 of 2 overloads)

Decrement operator (prefix).

```
buffers_iterator & operator--();
```

5.94.12.2 buffers_iterator::operator-- (2 of 2 overloads)

Decrement operator (postfix).

```
buffers_iterator operator--(int);
```

5.94.13 buffers_iterator::operator-=

Subtraction operator.

```
buffers_iterator & operator+=(std::ptrdiff_t difference);
```

5.94.14 buffers_iterator::operator->

Dereference an iterator.

```
pointer operator->() const;
```

5.94.15 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

5.94.16 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

5.94.17 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

5.94.18 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

5.94.19 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

5.94.20 buffers_iterator::operator[]

Access an individual element.

```
reference operator[](  
    std::ptrdiff_t difference) const;
```

5.94.21 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

5.94.22 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

5.94.23 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

5.95 connect

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename EndpointSequence>
Protocol::endpoint connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);

template<
    typename Protocol,
    typename EndpointSequence>
Protocol::endpoint connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    asio::error_code & ec,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);

template<
    typename Protocol,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);

template<
    typename Protocol,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    asio::error_code & ec,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);

template<
    typename Protocol,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    Iterator end);

template<
    typename Protocol,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    Iterator end,
    asio::error_code & ec);

template<
```

```

typename Protocol,
typename EndpointSequence,
typename ConnectCondition>
Protocol::endpoint connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    ConnectCondition connect_condition,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);

template<
    typename Protocol,
    typename EndpointSequence,
    typename ConnectCondition>
Protocol::endpoint connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    ConnectCondition connect_condition,
    asio::error_code & ec,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);

template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    ConnectCondition connect_condition,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);

template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    ConnectCondition connect_condition,
    asio::error_code & ec,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);

template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    Iterator end,
    ConnectCondition connect_condition);

template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol > & s,

```

```
Iterator begin,
Iterator end,
ConnectCondition connect_condition,
asio::error_code & ec);
```

Requirements

Header: asio/connect.hpp

Convenience header: asio.hpp

5.95.1 connect (1 of 12 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename EndpointSequence>
Protocol::endpoint connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);
```

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.

endpoints A sequence of endpoints.

Return Value

The successfully connected endpoint.

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_context);
tcp::resolver::query q("host", "service");
tcp::socket s(io_context);
asio::connect(s, r.resolve(q));
```

5.95.2 connect (2 of 12 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename EndpointSequence>
Protocol::endpoint connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    asio::error_code & ec,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);
```

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.

endpoints 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, the successfully connected endpoint. Otherwise, a default-constructed endpoint.

Example

```
tcp::resolver r(io_context);
tcp::resolver::query q("host", "service");
tcp::socket s(io_context);
asio::error_code ec;
asio::connect(s, r.resolve(q), ec);
if (ec)
{
    // An error occurred.
}
```

5.95.3 connect (3 of 12 overloads)

(Deprecated.) Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);
```

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`.

5.95.4 connect (4 of 12 overloads)

(Deprecated.) Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    asio::error_code & ec,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);
```

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`.

5.95.5 connect (5 of 12 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol > & 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

An iterator denoting the successfully connected endpoint.

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_context);
tcp::resolver::query q("host", "service");
tcp::resolver::results_type e = r.resolve(q);
tcp::socket s(io_context);
asio::connect(s, e.begin(), e.end());
```

5.95.6 connect (6 of 12 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename Iterator>
Iterator connect(
    basic_socket< Protocol > & 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_context);
tcp::resolver::query q("host", "service");
tcp::resolver::results_type e = r.resolve(q);
tcp::socket s(io_context);
asio::error_code ec;
asio::connect(s, e.begin(), e.end(), ec);
if (ec)
{
    // An error occurred.
}
```

5.95.7 connect (7 of 12 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename EndpointSequence,
    typename ConnectCondition>
Protocol::endpoint connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    ConnectCondition connect_condition,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);
```

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.
- endpoints** 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:

```
bool connect_condition(
    const asio::error_code& ec,
    const typename Protocol::endpoint& 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 the next endpoint to be tried. The function object should return true if the next endpoint should be tried, and false if it should be skipped.

Return Value

The successfully connected endpoint.

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
{
    bool operator()(const asio::error_code& ec,
                     const ::tcp::endpoint& next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next << std::endl;
        return true;
    }
};
```

It would be used with the `asio::connect` function as follows:

```
tcp::resolver r(io_context);
tcp::resolver::query q("host", "service");
tcp::socket s(io_context);
tcp::endpoint e = asio::connect(s,
    r.resolve(q), my_connect_condition());
std::cout << "Connected to: " << e << std::endl;
```

5.95.8 connect (8 of 12 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename EndpointSequence,
    typename ConnectCondition>
Protocol::endpoint connect(
    basic_socket< Protocol > & s,
    const EndpointSequence & endpoints,
    ConnectCondition connect_condition,
    asio::error_code & ec,
    typename enable_if< is_endpoint_sequence< EndpointSequence >::value >::type * = 0);
```

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.

endpoints 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:

```
bool connect_condition(
    const asio::error_code& ec,
    const typename Protocol::endpoint& 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 the next endpoint to be tried. The function object should return true if the next endpoint should be tried, and false if it should be skipped.

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, the successfully connected endpoint. Otherwise, a default-constructed endpoint.

Example

The following connect condition function object can be used to output information about the individual connection attempts:

```
struct my_connect_condition
{
    bool operator()(const asio::error_code& ec,
                     const ::tcp::endpoint& next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next << std::endl;
        return true;
    }
};
```

It would be used with the `asio::connect` function as follows:

```
tcp::resolver r(io_context);
tcp::resolver::query q("host", "service");
tcp::socket s(io_context);
asio::error_code ec;
tcp::endpoint e = asio::connect(s,
    r.resolve(q), my_connect_condition(), ec);
if (ec)
{
    // An error occurred.
}
else
{
    std::cout << "Connected to: " << e << std::endl;
}
```

5.95.9 connect (9 of 12 overloads)

(Deprecated.) Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    ConnectCondition connect_condition,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);
```

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:

```
bool connect_condition(
    const asio::error_code& ec,
    const typename Protocol::endpoint& 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 the next endpoint to be tried. The function object should return true if the next endpoint should be tried, and false if it should be skipped.

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`.

5.95.10 connect (10 of 12 overloads)

(Deprecated.) Establishes a socket connection by trying each endpoint in a sequence.

```

template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol > & s,
    Iterator begin,
    ConnectCondition connect_condition,
    asio::error_code & ec,
    typename enable_if<!is_endpoint_sequence< Iterator >::value >::type * = 0);

```

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:

```

bool connect_condition(
    const asio::error_code& ec,
    const typename Protocol::endpoint& 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 the next endpoint to be tried. The function object should return true if the next endpoint should be tried, and false if it should be skipped.

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`.

5.95.11 `connect` (11 of 12 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```

template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol > & 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:

```
bool connect_condition(
    const asio::error_code& ec,
    const typename Protocol::endpoint& 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 the next endpoint to be tried. The function object should return true if the next endpoint should be tried, and false if it should be skipped.

Return Value

An iterator denoting the successfully connected endpoint.

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
{
    bool operator()(const asio::error_code& ec,
                     const ::tcp::endpoint& next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next << std::endl;
        return true;
    }
};
```

It would be used with the `asio::connect` function as follows:

```
tcp::resolver r(io_context);
tcp::resolver::query q("host", "service");
tcp::resolver::results_type e = r.resolve(q);
tcp::socket s(io_context);
tcp::resolver::results_type::iterator i = asio::connect(
    s, e.begin(), e.end(), my_connect_condition());
std::cout << "Connected to: " << i->endpoint() << std::endl;
```

5.95.12 connect (12 of 12 overloads)

Establishes a socket connection by trying each endpoint in a sequence.

```
template<
    typename Protocol,
    typename Iterator,
    typename ConnectCondition>
Iterator connect(
    basic_socket< Protocol > & 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:

```
bool connect_condition(
    const asio::error_code& ec,
    const typename Protocol::endpoint& 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 the next endpoint to be tried. The function object should return true if the next endpoint should be tried, and false if it should be skipped.

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
{
    bool operator() (
        const asio::error_code& ec,
        const ::tcp::endpoint& next)
    {
        if (ec) std::cout << "Error: " << ec.message() << std::endl;
        std::cout << "Trying: " << next << std::endl;
        return true;
    }
};
```

It would be used with the `asio::connect` function as follows:

```
tcp::resolver r(io_context);
tcp::resolver::query q("host", "service");
tcp::resolver::results_type e = r.resolve(q);
tcp::socket s(io_context);
asio::error_code ec;
tcp::resolver::results_type::iterator i = asio::connect(
    s, e.begin(), e.end(), my_connect_condition());
if (ec)
{
    // An error occurred.
}
else
{
    std::cout << "Connected to: " << i->endpoint() << std::endl;
}
```

5.96 const_buffer

Holds a buffer that cannot be modified.

```
class const_buffer
```

Member Functions

Name	Description
<code>const_buffer</code>	Construct an empty buffer. Construct a buffer to represent a given memory range. Construct a non-modifiable buffer from a modifiable one.
<code>data</code>	Get a pointer to the beginning of the memory range.
<code>operator+=</code>	Move the start of the buffer by the specified number of bytes.
<code>size</code>	Get the size of the memory range.

Related Functions

Name	Description
<code>operator+</code>	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 `data()` and `size()` member functions:

```
asio::const_buffer b1 = ...;
std::size_t s1 = b1.size();
const unsigned char* p1 = static_cast<const unsigned char*>(b1.data());
```

The `data()` member 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`

5.96.1 `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);
```

5.96.1.1 `const_buffer::const_buffer (1 of 3 overloads)`

Construct an empty buffer.

```
const_buffer();
```

5.96.1.2 `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);
```

5.96.1.3 `const_buffer::const_buffer (3 of 3 overloads)`

Construct a non-modifiable buffer from a modifiable one.

```
const_buffer(
    const mutable_buffer & b);
```

5.96.2 const_buffer::data

Get a pointer to the beginning of the memory range.

```
const void * data() const;
```

5.96.3 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 n);

const_buffer operator+
    std::size_t n,
    const const_buffer & b);
```

5.96.3.1 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 n);
```

5.96.3.2 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 n,
    const const_buffer & b);
```

5.96.4 const_buffer::operator+=

Move the start of the buffer by the specified number of bytes.

```
const_buffer & operator+=(  
    std::size_t n);
```

5.96.5 const_buffer::size

Get the size of the memory range.

```
std::size_t size() const;
```

5.97 const_buffers_1

(Deprecated: Use [const_buffer](#).) 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
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.
const_buffers_1	Construct to represent a given memory range. Construct to represent a single non-modifiable buffer.
data	Get a pointer to the beginning of the memory range.
end	Get a random-access iterator for one past the last element.
operator+=	Move the start of the buffer by the specified number of bytes.
size	Get the size of the memory range.

Related Functions

Name	Description
operator+	Create a new non-modifiable buffer that is offset from the start of another.

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.97.1 const_buffers_1::begin

Get a random-access iterator to the first element.

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

5.97.2 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);
```

5.97.2.1 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);
```

5.97.2.2 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);
```

5.97.3 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

5.97.4 const_buffers_1::data

Inherited from const_buffer.

Get a pointer to the beginning of the memory range.

```
const void * data() const;
```

5.97.5 const_buffers_1::end

Get a random-access iterator for one past the last element.

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

5.97.6 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 n);

const_buffer operator+
    std::size_t n,
    const const_buffer & b);
```

5.97.6.1 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 n);
```

5.97.6.2 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 n,
    const const_buffer & b);
```

5.97.7 const_buffers_1::operator+=

Inherited from const_buffer.

Move the start of the buffer by the specified number of bytes.

```
const_buffer & operator+=(
    std::size_t n);
```

5.97.8 const_buffers_1::size

Inherited from const_buffer.

Get the size of the memory range.

```
std::size_t size() const;
```

5.97.9 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.
data	Get a pointer to the beginning of the memory range.
operator+=	Move the start of the buffer by the specified number of bytes.
size	Get the size of the memory range.

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 `data()` and `size()` member functions:

```
asio::const_buffer b1 = ...;
std::size_t s1 = b1.size();
const unsigned char* p1 = static_cast<const unsigned char*>(b1.data());
```

The `data()` member 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`

5.98 coroutine

Provides support for implementing stackless coroutines.

```
class coroutine
```

Member Functions

Name	Description
coroutine	Constructs a coroutine in its initial state.

Name	Description
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_context_));
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_context::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`

5.98.1 `coroutine::coroutine`

Constructs a coroutine in its initial state.

```
coroutine();
```

5.98.2 `coroutine::is_child`

Returns true if the coroutine is the child of a fork.

```
bool is_child() const;
```

5.98.3 `coroutine::is_complete`

Returns true if the coroutine has reached its terminal state.

```
bool is_complete() const;
```

5.98.4 coroutine::is_parent

Returns true if the coroutine is the parent of a fork.

```
bool is_parent() const;
```

5.99 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.
executor_type	The type of the executor associated with the object.
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. Move-construct a basic_deadline_timer from another.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.

Name	Description
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code> .) Get the <code>io_context</code> associated with the object.
<code>operator=</code>	Move-assign a <code>basic_deadline_timer</code> from another.
<code>wait</code>	Perform a blocking wait on the timer.
<code>~basic_deadline_timer</code>	Destroys the timer.

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_context);

// 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_context,
    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:

```
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`

5.100 defer

Submits a completion token or function object for execution.

```
template<
    typename CompletionToken>
DEDUCED defer(
    CompletionToken && token);

template<
    typename Executor,
    typename CompletionToken>
DEDUCED defer(
    const Executor & ex,
    CompletionToken && token,
    typename enable_if< is_executor< Executor >::value >::type * = 0);

template<
    typename ExecutionContext,
```

```

    typename CompletionToken>
DEDUCED defer(
    ExecutionContext & ctx,
    CompletionToken && token,
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >:: type * = 0);

```

Requirements

Header: asio/defer.hpp

Convenience header: asio.hpp

5.100.1 defer (1 of 3 overloads)

Submits a completion token or function object for execution.

```

template<
    typename CompletionToken>
DEDUCED defer(
    CompletionToken && token);

```

This function submits an object for execution using the object's associated executor. The function object is queued for execution, and is never called from the current thread prior to returning from `defer()`.

This function has the following effects:

- Constructs a function object handler of type `Handler`, initialized with `handler(forward<CompletionToken>(token))`.
- Constructs an object `result` of type `async_result<Handler>`, initializing the object as `result(handler)`.
- Obtains the handler's associated executor object `ex` by performing `get_associated_executor(handler)`.
- Obtains the handler's associated allocator object `alloc` by performing `get_associated_allocator(handler)`.
- Performs `ex.defer(std::move(handler), alloc)`.
- Returns `result.get()`.

5.100.2 defer (2 of 3 overloads)

Submits a completion token or function object for execution.

```

template<
    typename Executor,
    typename CompletionToken>
DEDUCED defer(
    const Executor & ex,
    CompletionToken && token,
    typename enable_if< is_executor< Executor >::value >::type * = 0);

```

This function submits an object for execution using the specified executor. The function object is queued for execution, and is never called from the current thread prior to returning from `defer()`.

This function has the following effects:

- Constructs a function object handler of type `Handler`, initialized with `handler(forward<CompletionToken>(token))`.

- Constructs an object `result` of type `async_result<Handler>`, initializing the object as `result(handler)`.
- Obtains the handler's associated executor object `ex1` by performing `get_associated_executor(handler)`.
- Creates a work object `w` by performing `make_work(ex1)`.
- Obtains the handler's associated allocator object `alloc` by performing `get_associated_allocator(handler)`.
- Constructs a function object `f` with a function call operator that performs `ex1.dispatch(std::move(handler), alloc)` followed by `w.reset()`.
- Performs `Executor(ex).defer(std::move(f), alloc)`.
- Returns `result.get()`.

5.100.3 defer (3 of 3 overloads)

Submits a completion token or function object for execution.

```
template<
    typename ExecutionContext,
    typename CompletionToken>
DEDUCED defer(
    ExecutionContext & ctx,
    CompletionToken && token,
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::type * = 0);
```

Return Value

```
defer(ctx.get_executor(), forward<CompletionToken>(token)).
```

5.101 dispatch

Submits a completion token or function object for execution.

```
template<
    typename CompletionToken>
DEDUCED dispatch(
    CompletionToken && token);

template<
    typename Executor,
    typename CompletionToken>
DEDUCED dispatch(
    const Executor & ex,
    CompletionToken && token,
    typename enable_if< is_executor< Executor >::value >::type * = 0);

template<
    typename ExecutionContext,
    typename CompletionToken>
DEDUCED dispatch(
    ExecutionContext & ctx,
    CompletionToken && token,
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::type * = 0);
```

Requirements

Header: asio/dispatch.hpp

Convenience header: asio.hpp

5.101.1 dispatch (1 of 3 overloads)

Submits a completion token or function object for execution.

```
template<
    typename CompletionToken>
DEDUCED dispatch(
    CompletionToken && token);
```

This function submits an object for execution using the object's associated executor. The function object is queued for execution, and is never called from the current thread prior to returning from `dispatch()`.

This function has the following effects:

- Constructs a function object handler of type `Handler`, initialized with `handler(forward<CompletionToken>(token))`.
- Constructs an object `result` of type `async_result<Handler>`, initializing the object as `result(handler)`.
- Obtains the handler's associated executor object `ex` by performing `get_associated_executor(handler)`.
- Obtains the handler's associated allocator object `alloc` by performing `get_associated_allocator(handler)`.
- Performs `ex.dispatch(std::move(handler), alloc)`.
- Returns `result.get()`.

5.101.2 dispatch (2 of 3 overloads)

Submits a completion token or function object for execution.

```
template<
    typename Executor,
    typename CompletionToken>
DEDUCED dispatch(
    const Executor & ex,
    CompletionToken && token,
    typename enable_if< is_executor< Executor >::value >::type * = 0);
```

This function submits an object for execution using the specified executor. The function object is queued for execution, and is never called from the current thread prior to returning from `dispatch()`.

This function has the following effects:

- Constructs a function object handler of type `Handler`, initialized with `handler(forward<CompletionToken>(token))`.
- Constructs an object `result` of type `async_result<Handler>`, initializing the object as `result(handler)`.
- Obtains the handler's associated executor object `ex1` by performing `get_associated_executor(handler)`.
- Creates a work object `w` by performing `make_work(ex1)`.
- Obtains the handler's associated allocator object `alloc` by performing `get_associated_allocator(handler)`.
- Constructs a function object `f` with a function call operator that performs `ex1.dispatch(std::move(handler), alloc)` followed by `w.reset()`.
- Performs `Executor(ex).dispatch(std::move(f), alloc)`.
- Returns `result.get()`.

5.101.3 dispatch (3 of 3 overloads)

Submits a completion token or function object for execution.

```
template<
    typename ExecutionContext,
    typename CompletionToken>
DEDUCED dispatch(
    ExecutionContext & ctx,
    CompletionToken && token,
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::type * = 0);
```

Return Value

```
dispatch(ctx.get_executor(), forward<CompletionToken>(token)).
```

5.102 dynamic_buffer

The `asio::dynamic_buffer` function is used to create a dynamically resized buffer from a `std::basic_string` or `std::vector`.

```
template<
    typename Elem,
    typename Traits,
    typename Allocator>
dynamic_string_buffer< Elem, Traits, Allocator > dynamic_buffer(
    std::basic_string< Elem, Traits, Allocator > & data);

template<
    typename Elem,
    typename Traits,
    typename Allocator>
dynamic_string_buffer< Elem, Traits, Allocator > dynamic_buffer(
    std::basic_string< Elem, Traits, Allocator > & data,
    std::size_t max_size);

template<
    typename Elem,
    typename Allocator>
dynamic_vector_buffer< Elem, Allocator > dynamic_buffer(
    std::vector< Elem, Allocator > & data);

template<
    typename Elem,
    typename Allocator>
dynamic_vector_buffer< Elem, Allocator > dynamic_buffer(
    std::vector< Elem, Allocator > & data,
    std::size_t max_size);
```

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.102.1 dynamic_buffer (1 of 4 overloads)

Create a new dynamic buffer that represents the given string.

```
template<
    typename Elem,
    typename Traits,
    typename Allocator>
dynamic_string_buffer< Elem, Traits, Allocator > dynamic_buffer(
    std::basic_string< Elem, Traits, Allocator > & data);
```

Return Value

```
dynamic_string_buffer<Elem, Traits, Allocator>(data).
```

5.102.2 dynamic_buffer (2 of 4 overloads)

Create a new dynamic buffer that represents the given string.

```
template<
    typename Elem,
    typename Traits,
    typename Allocator>
dynamic_string_buffer< Elem, Traits, Allocator > dynamic_buffer(
    std::basic_string< Elem, Traits, Allocator > & data,
    std::size_t max_size);
```

Return Value

```
dynamic_string_buffer<Elem, Traits, Allocator>(data, max_size).
```

5.102.3 dynamic_buffer (3 of 4 overloads)

Create a new dynamic buffer that represents the given vector.

```
template<
    typename Elem,
    typename Allocator>
dynamic_vector_buffer< Elem, Allocator > dynamic_buffer(
    std::vector< Elem, Allocator > & data);
```

Return Value

```
dynamic_vector_buffer<Elem, Allocator>(data).
```

5.102.4 dynamic_buffer (4 of 4 overloads)

Create a new dynamic buffer that represents the given vector.

```
template<
    typename Elem,
    typename Allocator>
dynamic_vector_buffer< Elem, Allocator > dynamic_buffer(
    std::vector< Elem, Allocator > & data,
    std::size_t max_size);
```

Return Value

```
dynamic_vector_buffer<Elem, Allocator>(data, max_size).
```

5.103 dynamic_string_buffer

Adapt a basic_string to the DynamicBuffer requirements.

```
template<
    typename Elem,
    typename Traits,
    typename Allocator>
class dynamic_string_buffer
```

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
capacity	Get the current capacity of the dynamic buffer.
commit	Move bytes from the output sequence to the input sequence.
consume	Remove characters from the input sequence.
data	Get a list of buffers that represents the input sequence.
dynamic_string_buffer	Construct a dynamic buffer from a string. Move construct a dynamic buffer.
max_size	Get the maximum size of the dynamic buffer.
prepare	Get a list of buffers that represents the output sequence, with the given size.
size	Get the size of the input sequence.

Requires that `sizeof(Elem) == 1`.

Requirements

Header: `asio/buffer.hpp`

Convenience header: asio.hpp

5.103.1 dynamic_string_buffer::capacity

Get the current capacity of the dynamic buffer.

```
std::size_t capacity() const;
```

Return Value

The current total capacity of the buffer, i.e. for both the input sequence and output sequence.

5.103.2 dynamic_string_buffer::commit

Move bytes from the output sequence to the input sequence.

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

Parameters

- The number of bytes to append from the start of the output sequence to the end of the input sequence. The remainder of the output sequence is discarded.

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.

5.103.3 dynamic_string_buffer::const_buffers_type

The type used to represent the input sequence as a list of buffers.

```
typedef const_buffer const_buffers_type;
```

Member Functions

Name	Description
<code>const_buffer</code>	Construct an empty buffer. Construct a buffer to represent a given memory range. Construct a non-modifiable buffer from a modifiable one.
<code>data</code>	Get a pointer to the beginning of the memory range.
<code>operator+=</code>	Move the start of the buffer by the specified number of bytes.
<code>size</code>	Get the size of the memory range.

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 `data()` and `size()` member functions:

```
asio::const_buffer b1 = ...;
std::size_t s1 = b1.size();
const unsigned char* p1 = static_cast<const unsigned char*>(b1.data());
```

The `data()` member 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`

5.103.4 `dynamic_string_buffer::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.

5.103.5 `dynamic_string_buffer::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 the `basic_string` memory in input sequence.

Remarks

The returned object is invalidated by any `dynamic_string_buffer` or `basic_string` member function that modifies the input sequence or output sequence.

5.103.6 `dynamic_string_buffer::dynamic_string_buffer`

Construct a dynamic buffer from a string.

```
explicit dynamic_string_buffer(
    std::basic_string< Elem, Traits, Allocator > & s,
    std::size_t maximum_size = (std::numeric_limits< std::size_t >::max)());
```

Move construct a dynamic buffer.

```
dynamic_string_buffer(
    dynamic_string_buffer && other);
```

5.103.6.1 `dynamic_string_buffer::dynamic_string_buffer (1 of 2 overloads)`

Construct a dynamic buffer from a string.

```
dynamic_string_buffer(
    std::basic_string< Elem, Traits, Allocator > & s,
    std::size_t maximum_size = (std::numeric_limits< std::size_t >::max)());
```

Parameters

- s The string to be used as backing storage for the dynamic buffer. Any existing data in the string is treated as the dynamic buffer's input sequence. The object stores a reference to the string and the user is responsible for ensuring that the string object remains valid until the `dynamic_string_buffer` object is destroyed.

maximum_size Specifies a maximum size for the buffer, in bytes.

5.103.6.2 `dynamic_string_buffer::dynamic_string_buffer (2 of 2 overloads)`

Move construct a dynamic buffer.

```
dynamic_string_buffer(
    dynamic_string_buffer && other);
```

5.103.7 `dynamic_string_buffer::max_size`

Get the maximum size of the dynamic buffer.

```
std::size_t max_size() const;
```

Return Value

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

5.103.8 `dynamic_string_buffer::mutable_buffers_type`

The type used to represent the output sequence as a list of buffers.

```
typedef mutable_buffer mutable_buffers_type;
```

Member Functions

Name	Description
<code>data</code>	Get a pointer to the beginning of the memory range.
<code>mutable_buffer</code>	Construct an empty buffer. Construct a buffer to represent a given memory range.
<code>operator+=</code>	Move the start of the buffer by the specified number of bytes.
<code>size</code>	Get the size of the 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 `data()` and `size()` member functions:

```
asio::mutable_buffer b1 = ...;
std::size_t s1 = b1.size();
unsigned char* p1 = static_cast<unsigned char*>(b1.data());
```

The `data()` member 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`

5.103.9 `dynamic_string_buffer::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` bytes, resizing the `basic_string` object as necessary.

Return Value

An object of type `mutable_buffers_type` that satisfies MutableBufferSequence requirements, representing `basic_string` memory at the start of the output sequence of size `n`.

Exceptions

`std::length_error` If `size() + n > max_size()`.

Remarks

The returned object is invalidated by any `dynamic_string_buffer` or `basic_string` member function that modifies the input sequence or output sequence.

5.103.10 `dynamic_string_buffer::size`

Get the size of the input sequence.

```
std::size_t size() const;
```

5.104 `dynamic_vector_buffer`

Adapt a vector to the DynamicBuffer requirements.

```
template<
    typename Elem,
    typename Allocator>
class dynamic_vector_buffer
```

Types

Name	Description
<code>const_buffers_type</code>	The type used to represent the input sequence as a list of buffers.
<code>mutable_buffers_type</code>	The type used to represent the output sequence as a list of buffers.

Member Functions

Name	Description
<code>capacity</code>	Get the current capacity of the dynamic buffer.
<code>commit</code>	Move bytes from the output sequence to the input sequence.
<code>consume</code>	Remove characters from the input sequence.

Name	Description
data	Get a list of buffers that represents the input sequence.
dynamic_vector_buffer	Construct a dynamic buffer from a string. Move construct a dynamic buffer.
max_size	Get the maximum size of the dynamic buffer.
prepare	Get a list of buffers that represents the output sequence, with the given size.
size	Get the size of the input sequence.

Requires that `sizeof(Elem) == 1`.

Requirements

Header: `asio/buffer.hpp`

Convenience header: `asio.hpp`

5.104.1 `dynamic_vector_buffer::capacity`

Get the current capacity of the dynamic buffer.

```
std::size_t capacity() const;
```

Return Value

The current total capacity of the buffer, i.e. for both the input sequence and output sequence.

5.104.2 `dynamic_vector_buffer::commit`

Move bytes from the output sequence to the input sequence.

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

Parameters

- The number of bytes to append from the start of the output sequence to the end of the input sequence. The remainder of the output sequence is discarded.

Requires a preceding call `prepare(x)` where $x \geq n$, 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.

5.104.3 dynamic_vector_buffer::const_buffers_type

The type used to represent the input sequence as a list of buffers.

```
typedef const_buffer const_buffers_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.
data	Get a pointer to the beginning of the memory range.
operator+=	Move the start of the buffer by the specified number of bytes.
size	Get the size of the memory range.

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 `data()` and `size()` member functions:

```
asio::const_buffer b1 = ...;
std::size_t s1 = b1.size();
const unsigned char* p1 = static_cast<const unsigned char*>(b1.data());
```

The `data()` member 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`

5.104.4 dynamic_vector_buffer::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.

5.104.5 `dynamic_vector_buffer::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 the `basic_string` memory in input sequence.

Remarks

The returned object is invalidated by any `dynamic_vector_buffer` or `basic_string` member function that modifies the input sequence or output sequence.

5.104.6 `dynamic_vector_buffer::dynamic_vector_buffer`

Construct a dynamic buffer from a string.

```
explicit dynamic_vector_buffer(
    std::vector< Elem, Allocator > & v,
    std::size_t maximum_size = (std::numeric_limits< std::size_t >::max)());
```

Move construct a dynamic buffer.

```
dynamic_vector_buffer(
    dynamic_vector_buffer && other);
```

5.104.6.1 `dynamic_vector_buffer::dynamic_vector_buffer (1 of 2 overloads)`

Construct a dynamic buffer from a string.

```
dynamic_vector_buffer(
    std::vector< Elem, Allocator > & v,
    std::size_t maximum_size = (std::numeric_limits< std::size_t >::max)());
```

Parameters

- ▀ The vector to be used as backing storage for the dynamic buffer. Any existing data in the vector is treated as the dynamic buffer's input sequence. The object stores a reference to the vector and the user is responsible for ensuring that the vector object remains valid until the `dynamic_vector_buffer` object is destroyed.

maximum_size Specifies a maximum size for the buffer, in bytes.

5.104.6.2 `dynamic_vector_buffer::dynamic_vector_buffer (2 of 2 overloads)`

Move construct a dynamic buffer.

```
dynamic_vector_buffer(
    dynamic_vector_buffer && other);
```

5.104.7 `dynamic_vector_buffer::max_size`

Get the maximum size of the dynamic buffer.

```
std::size_t max_size() const;
```

Return Value

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

5.104.8 `dynamic_vector_buffer::mutable_buffers_type`

The type used to represent the output sequence as a list of buffers.

```
typedef mutable_buffer mutable_buffers_type;
```

Member Functions

Name	Description
data	Get a pointer to the beginning of the memory range.
mutable_buffer	Construct an empty buffer. Construct a buffer to represent a given memory range.
operator+=	Move the start of the buffer by the specified number of bytes.
size	Get the size of the 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 `data()` and `size()` member functions:

```
asio::mutable_buffer b1 = ...;
std::size_t s1 = b1.size();
unsigned char* p1 = static_cast<unsigned char*>(b1.data());
```

The `data()` member 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`

5.104.9 `dynamic_vector_buffer::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` bytes, resizing the `basic_string` object as necessary.

Return Value

An object of type `mutable_buffers_type` that satisfies `MutableBufferSequence` requirements, representing `basic_string` memory at the start of the output sequence of size `n`.

Exceptions

`std::length_error` If `size() + n > max_size()`.

Remarks

The returned object is invalidated by any `dynamic_vector_buffer` or `basic_string` member function that modifies the input sequence or output sequence.

5.104.10 `dynamic_vector_buffer::size`

Get the size of the input sequence.

```
std::size_t size() const;
```

5.105 `error::addrinfo_category`

```
static const asio::error_category & addrinfo_category = asio::error::get_addrinfo_category();
```

Requirements

Header: `asio/error.hpp`

Convenience header: `asio.hpp`

5.106 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

5.107 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

5.108 error::get_addrinfo_category

```
const asio::error_category & get_addrinfo_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.109 error::get_misc_category

```
const asio::error_category & get_misc_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.110 error::get_netdb_category

```
const asio::error_category & get_netdb_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.111 error::get_ssl_category

```
const asio::error_category & get_ssl_category();
```

Requirements

Header: asio/ssl/error.hpp

Convenience header: asio/ssl.hpp

5.112 error::get_system_category

```
const asio::error_category & get_system_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.113 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

5.113.1 error::make_error_code (1 of 5 overloads)

```
asio::error_code make_error_code(  
    basic_errors e);
```

5.113.2 error::make_error_code (2 of 5 overloads)

```
asio::error_code make_error_code(  
    netdb_errors e);
```

5.113.3 error::make_error_code (3 of 5 overloads)

```
asio::error_code make_error_code(  
    addrinfo_errors e);
```

5.113.4 error::make_error_code (4 of 5 overloads)

```
asio::error_code make_error_code(  
    misc_errors e);
```

5.113.5 error::make_error_code (5 of 5 overloads)

```
asio::error_code make_error_code(  
    ssl_errors e);
```

5.114 error::misc_category

```
static const asio::error_category & misc_category = asio::error::get_misc_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.115 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

5.116 error::netdb_category

```
static const asio::error_category & netdb_category = asio::error::get_netdb_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.117 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

5.118 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

5.119 error::ssl_errors

```
enum ssl_errors
```

Requirements

Header: asio/ssl/error.hpp

Convenience header: asio/ssl.hpp

5.120 error::system_category

```
static const asio::error_category & system_category = asio::error::get_system_category();
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.121 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

5.121.1 error_category::message

Returns a string describing the error denoted by `value`.

```
std::string message(  
    int value) const;
```

5.121.2 error_category::name

Returns a string naming the error category.

```
const char * name() const;
```

5.121.3 error_category::operator!=

Inequality operator to compare two error categories.

```
bool operator!=(  
    const error_category & rhs) const;
```

5.121.4 error_category::operator==

Equality operator to compare two error categories.

```
bool operator==(  
    const error_category & rhs) const;
```

5.121.5 error_category::~error_category

Destructor.

```
virtual ~error_category();
```

5.122 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
assign	Assign a new error value.
category	Get the error category.
clear	Clear the error value to the default.
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

5.122.1 error_code::assign

Assign a new error value.

```
void assign(
    int v,
    const error_category & c);
```

5.122.2 error_code::category

Get the error category.

```
const error_category & category() const;
```

5.122.3 error_code::clear

Clear the error value to the default.

```
void clear();
```

5.122.4 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);
```

5.122.4.1 error_code::error_code (1 of 3 overloads)

Default constructor.

```
error_code();
```

5.122.4.2 error_code::error_code (2 of 3 overloads)

Construct with specific error code and category.

```
error_code(
    int v,
    const error_category & c);
```

5.122.4.3 error_code::error_code (3 of 3 overloads)

Construct from an error code enum.

```
template<
    typename ErrorEnum>
error_code(
    ErrorEnum e);
```

5.122.5 error_code::message

Get the message associated with the error.

```
std::string message() const;
```

5.122.6 `error_code::operator unspecified_bool_type`

Operator returns non-null if there is a non-success error code.

```
operator unspecified_bool_type() const;
```

5.122.7 `error_code::operator!`

Operator to test if the error represents success.

```
bool operator!() const;
```

5.122.8 `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

5.122.9 `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

5.122.10 `error_code::unspecified_bool_true`

```
static void unspecified_bool_true(
    unspecified_bool_type_t );
```

5.122.11 `error_code::unspecified_bool_type`

```
typedef void(*) unspecified_bool_type;
```

Requirements

Header: asio/error_code.hpp

Convenience header: asio.hpp

5.122.12 error_code::value

Get the error value.

```
int value() const;
```

5.123 error_code::unspecified_bool_type_t

```
struct unspecified_bool_type_t
```

Requirements

Header: asio/error_code.hpp

Convenience header: asio.hpp

5.124 execution_context

A context for function object execution.

```
class execution_context :  
    noncopyable
```

Types

Name	Description
id	Class used to uniquely identify a service.
service	Base class for all io_context services.
fork_event	Fork-related event notifications.

Member Functions

Name	Description
notify_fork	Notify the execution_context of a fork-related event.

Protected Member Functions

Name	Description
destroy	Destroys all services in the context.
execution_context	Constructor.
shutdown	Shuts down all services in the context.
~execution_context	Destructor.

Friends

Name	Description
add_service	(Deprecated: Use make_service().) Add a service object to the execution_context.
has_service	Determine if an execution_context contains a specified service type.
make_service	Creates a service object and adds it to the execution_context.
use_service	Obtain the service object corresponding to the given type.

An execution context represents a place where function objects will be executed. An `io_context` is an example of an execution context.

The `execution_context` class and services

Class `execution_context` implements an extensible, type-safe, polymorphic set of services, indexed by service type.

Services exist to manage the resources that are shared across an execution context. For example, timers may be implemented in terms of a single timer queue, and this queue would be stored in a service.

Access to the services of an `execution_context` 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 `execution_context`, an object of type `Service` is created and added to the `execution_context`. A C++ program can check if an `execution_context` implements a particular service with the function template `has_service<Service>()`.

Service objects may be explicitly added to an `execution_context` 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 `execution_context` parameter, the `invalid_service_owner` exception is thrown.

Once a service reference is obtained from an `execution_context` object by calling `use_service()`, that reference remains usable as long as the owning `execution_context` object exists.

All service implementations have `execution_context::service` as a public base class. Custom services may be implemented by deriving from this class and then added to an `execution_context` using the facilities described above.

The `execution_context` as a base class

Class `execution_context` may be used only as a base class for concrete execution context types. The `io_context` is an example of such a derived type.

On destruction, a class that is derived from `execution_context` must perform `execution_context::shutdown()` followed by `execution_context::destroy()`.

This destruction sequence 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_context` function `stop()` is called to terminate any `run()` calls as soon as possible. The `io_context` destructor calls `shutdown()` and `destroy()` to destroy all pending handlers, causing all `shared_ptr` references to all connection objects to be destroyed.

Requirements

Header: `asio/execution_context.hpp`

Convenience header: `asio.hpp`

5.124.1 `execution_context::add_service`

(Deprecated: Use `make_service()`.) Add a service object to the `execution_context`.

```
template<
    typename Service>
friend void add_service(
    execution_context & e,
    Service * svc);
```

This function is used to add a service to the `execution_context`.

Parameters

`e` The `execution_context` object that owns the service.

`svc` The service object. On success, ownership of the service object is transferred to the `execution_context`. When the `execution_context` object is destroyed, it will destroy the service object by performing:

```
delete static_cast<execution_context::service*>(svc)
```

Exceptions

`asio::service_already_exists` Thrown if a service of the given type is already present in the `execution_context`.

`asio::invalid_service_owner` Thrown if the service's owning `execution_context` is not the `execution_context` object specified by the `e` parameter.

Requirements

Header: `asio/execution_context.hpp`

Convenience header: `asio.hpp`

5.124.2 execution_context::destroy

Destroys all services in the context.

```
void destroy();
```

This function is implemented as follows:

- For each service object `svc` in the `execution_context` set, in reverse order * of the beginning of service object lifetime, performs
`delete static_cast<execution_context::service*>(svc).`

5.124.3 execution_context::execution_context

Constructor.

```
execution_context();
```

5.124.4 execution_context::fork_event

Fork-related event notifications.

```
enum fork_event
```

Values

fork_prepare Notify the context that the process is about to fork.

fork_parent Notify the context that the process has forked and is the parent.

fork_child Notify the context that the process has forked and is the child.

5.124.5 execution_context::has_service

Determine if an `execution_context` contains a specified service type.

```
template<
    typename Service>
friend bool has_service(
    execution_context & e);
```

This function is used to determine whether the `execution_context` contains a service object corresponding to the given service type.

Parameters

e The `execution_context` object that owns the service.

Return Value

A boolean indicating whether the `execution_context` contains the service.

Requirements

Header: asio/execution_context.hpp

Convenience header: asio.hpp

5.124.6 execution_context::make_service

Creates a service object and adds it to the [execution_context](#).

```
template<
    typename Service,
    typename... Args>
friend Service & make_service(
    execution_context & e,
    Args &&... args);
```

This function is used to add a service to the [execution_context](#).

Parameters

e The [execution_context](#) object that owns the service.

args Zero or more arguments to be passed to the service constructor.

Exceptions

asio::service_already_exists Thrown if a service of the given type is already present in the [execution_context](#).

Requirements

Header: asio/execution_context.hpp

Convenience header: asio.hpp

5.124.7 execution_context::notify_fork

Notify the [execution_context](#) of a fork-related event.

```
void notify_fork(
    fork_event event);
```

This function is used to inform the [execution_context](#) that the process is about to fork, or has just forked. This allows the [execution_context](#), 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 [execution_context](#) function, or any function associated with the [execution_context](#)'s derived class, 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 [execution_context](#) or its derived class.

Parameters

event A fork-related event.

Exceptions

asio::system_error Thrown on failure. If the notification fails the `execution_context` object should no longer be used and should be destroyed.

Example

The following code illustrates how to incorporate the `notify_fork()` function:

```
my_execution_context.notify_fork(execution_context::fork_prepare);
if (fork() == 0)
{
    // This is the child process.
    my_execution_context.notify_fork(execution_context::fork_child);
}
else
{
    // This is the parent process.
    my_execution_context.notify_fork(execution_context::fork_parent);
}
```

Remarks

For each service object `svc` in the `execution_context` set, performs `svc->notify_fork();`. 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.

5.124.8 execution_context::shutdown

Shuts down all services in the context.

```
void shutdown();
```

This function is implemented as follows:

- For each service object `svc` in the `execution_context` set, in reverse order of the beginning of service object lifetime, performs `svc->shutdown();`.

5.124.9 execution_context::use_service

Obtain the service object corresponding to the given type.

```
template<
    typename Service>
friend Service & use_service(
    execution_context & e);

template<
    typename Service>
friend Service & use_service(
    io_context & ioc);
```

5.124.9.1 `execution_context::use_service` (1 of 2 overloads)

Obtain the service object corresponding to the given type.

```
template<
    typename Service>
friend Service & use_service(
    execution_context & e);
```

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 `execution_context` will create a new instance of the service.

Parameters

e The `execution_context` 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/execution_context.hpp`

Convenience header: `asio.hpp`

5.124.9.2 `execution_context::use_service` (2 of 2 overloads)

Obtain the service object corresponding to the given type.

```
template<
    typename Service>
friend Service & use_service(
    io_context & ioc);
```

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_context` will create a new instance of the service.

Parameters

ioc The `io_context` 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.

Remarks

This overload is preserved for backwards compatibility with services that inherit from `io_context::service`.

Requirements

Header: `asio/execution_context.hpp`

Convenience header: `asio.hpp`

5.124.10 execution_context::~execution_context

Destructor.

```
~execution_context();
```

5.125 execution_context::id

Class used to uniquely identify a service.

```
class id :  
    noncopyable
```

Member Functions

Name	Description
id	Constructor.

Requirements

Header: asio/execution_context.hpp

Convenience header: asio.hpp

5.125.1 execution_context::id::id

Constructor.

```
id();
```

5.126 execution_context::service

Base class for all **io_context** services.

```
class service :  
    noncopyable
```

Member Functions

Name	Description
context	Get the context object that owns the service.

Protected Member Functions

Name	Description
service	Constructor.
~service	Destructor.

Private Member Functions

Name	Description
notify_fork	Handle notification of a fork-related event to perform any necessary housekeeping.
shutdown	Destroy all user-defined handler objects owned by the service.

Requirements

Header: asio/execution_context.hpp

Convenience header: asio.hpp

5.126.1 execution_context::service::context

Get the context object that owns the service.

```
execution_context & context();
```

5.126.2 execution_context::service::service

Constructor.

```
service(
    execution_context & owner);
```

Parameters

owner The [execution_context](#) object that owns the service.

5.126.3 execution_context::service::~service

Destructor.

```
virtual ~service();
```

5.126.4 execution_context::service::notify_fork

Handle notification of a fork-related event to perform any necessary housekeeping.

```
virtual void notify_fork(  
    execution_context::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.

5.126.5 execution_context::service::shutdown

Destroy all user-defined handler objects owned by the service.

```
void shutdown();
```

5.127 executor

Polymorphic wrapper for executors.

```
class executor
```

Types

Name	Description
unspecified_bool_type_t	
unspecified_bool_type	

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the executor to invoke the given function object.
dispatch	Request the executor to invoke the given function object.
executor	Default constructor. Construct from nullptr. Copy constructor. Move constructor. Construct a polymorphic wrapper for the specified executor. Allocator-aware constructor to create a polymorphic wrapper for the specified executor.
on_work_finished	Inform the executor that some work is no longer outstanding.
on_work_started	Inform the executor that it has some outstanding work to do.
operator unspecified_bool_type	Operator to test if the executor contains a valid target.

Name	Description
operator=	Assignment operator. Assignment operator for nullptr_t. Assignment operator to create a polymorphic wrapper for the specified executor.
post	Request the executor to invoke the given function object.
target	Obtain a pointer to the target executor object.
target_type	Obtain type information for the target executor object.
unspecified_bool_true	
~executor	Destructor.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/executor.hpp

Convenience header: asio.hpp

5.127.1 executor::context

Obtain the underlying execution context.

```
execution_context & context() const;
```

5.127.2 executor::defer

Request the executor to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void defer(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the executor to execute the given function object. The function object is executed according to the rules of the target executor object.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.127.3 executor::dispatch

Request the executor to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void dispatch(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the executor to execute the given function object. The function object is executed according to the rules of the target executor object.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.127.4 executor::executor

Default constructor.

```
executor();
```

Construct from nullptr.

```
executor(
    nullptr_t );
```

Copy constructor.

```
executor(
    const executor & other);
```

Move constructor.

```
executor(
    executor && other);
```

Construct a polymorphic wrapper for the specified executor.

```
template<
    typename Executor>
executor(
    Executor e);
```

Allocator-aware constructor to create a polymorphic wrapper for the specified executor.

```
template<
    typename Executor,
    typename Allocator>
executor(
    allocator_arg_t ,
    const Allocator & a,
    Executor e);
```

5.127.4.1 executor::executor (1 of 6 overloads)

Default constructor.

```
executor();
```

5.127.4.2 executor::executor (2 of 6 overloads)

Construct from nullptr.

```
executor(
    nullptr_t );
```

5.127.4.3 executor::executor (3 of 6 overloads)

Copy constructor.

```
executor(
    const executor & other);
```

5.127.4.4 executor::executor (4 of 6 overloads)

Move constructor.

```
executor(
    executor && other);
```

5.127.4.5 executor::executor (5 of 6 overloads)

Construct a polymorphic wrapper for the specified executor.

```
template<
    typename Executor>
executor(
    Executor e);
```

5.127.4.6 executor::executor (6 of 6 overloads)

Allocator-aware constructor to create a polymorphic wrapper for the specified executor.

```
template<
    typename Executor,
    typename Allocator>
executor(
    allocator_arg_t ,
    const Allocator & a,
    Executor e);
```

5.127.5 executor::on_work_finished

Inform the executor that some work is no longer outstanding.

```
void on_work_finished() const;
```

5.127.6 executor::on_work_started

Inform the executor that it has some outstanding work to do.

```
void on_work_started() const;
```

5.127.7 executor::operator unspecified_bool_type

Operator to test if the executor contains a valid target.

```
operator unspecified_bool_type() const;
```

5.127.8 executor::operator!=

Compare two executors for inequality.

```
friend bool operator!=(
    const executor & a,
    const executor & b);
```

Requirements

Header: asio/executor.hpp

Convenience header: asio.hpp

5.127.9 executor::operator=

Assignment operator.

```
executor & operator=(
    const executor & other);
```

```
executor & operator=(
    executor && other);
```

Assignment operator for nullptr_t.

```
executor & operator=(  
    nullptr_t );
```

Assignment operator to create a polymorphic wrapper for the specified executor.

```
template<  
    typename Executor>  
executor & operator=(  
    Executor && e);
```

5.127.9.1 executor::operator= (1 of 4 overloads)

Assignment operator.

```
executor & operator=(  
    const executor & other);
```

5.127.9.2 executor::operator= (2 of 4 overloads)

```
executor & operator=(  
    executor && other);
```

5.127.9.3 executor::operator= (3 of 4 overloads)

Assignment operator for nullptr_t.

```
executor & operator=(  
    nullptr_t );
```

5.127.9.4 executor::operator= (4 of 4 overloads)

Assignment operator to create a polymorphic wrapper for the specified executor.

```
template<  
    typename Executor>  
executor & operator=(  
    Executor && e);
```

5.127.10 executor::operator==

Compare two executors for equality.

```
friend bool operator==(  
    const executor & a,  
    const executor & b);
```

Requirements

Header: asio/executor.hpp

Convenience header: asio.hpp

5.127.11 executor::post

Request the executor to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void post(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the executor to execute the given function object. The function object is executed according to the rules of the target executor object.

Parameters

- f** The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a** An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.127.12 executor::target

Obtain a pointer to the target executor object.

```
template<
    typename Executor>
Executor * target();
```



```
template<
    typename Executor>
const Executor * target() const;
```

5.127.12.1 executor::target (1 of 2 overloads)

Obtain a pointer to the target executor object.

```
template<
    typename Executor>
Executor * target();
```

Return Value

If `target_type() == typeid(T)`, a pointer to the stored executor target; otherwise, a null pointer.

5.127.12.2 executor::target (2 of 2 overloads)

Obtain a pointer to the target executor object.

```
template<
    typename Executor>
const Executor * target() const;
```

Return Value

If `target_type()` == `typeid(T)`, a pointer to the stored executor target; otherwise, a null pointer.

5.127.13 `executor::target_type`

Obtain type information for the target executor object.

```
const std::type_info & target_type() const;
```

Return Value

If `*this` has a target type of type `T`, `typeid(T)`; otherwise, `typeid(void)`.

5.127.14 `executor::unspecified_bool_true`

```
static void unspecified_bool_true(
    unspecified_bool_type_t );
```

5.127.15 `executor::unspecified_bool_type`

```
typedef void(*) unspecified_bool_type;
```

Requirements

Header: `asio/executor.hpp`

Convenience header: `asio.hpp`

5.127.16 `executor::~executor`

Destructor.

```
~executor();
```

5.128 `executor::unspecified_bool_type_t`

```
struct unspecified_bool_type_t
```

Requirements

Header: `asio/executor.hpp`

Convenience header: `asio.hpp`

5.129 `executor_arg`

A special value, similar to `std::nothrow`, used to disambiguate constructors that accept executor arguments.

```
constexpr executor_arg_t executor_arg;
```

See `executor_arg_t` and `uses_executor` for more information.

Requirements

Header: asio/uses_executor.hpp

Convenience header: asio.hpp

5.130 executor_arg_t

A special type, similar to std::nothrow_t, used to disambiguate constructors that accept executor arguments.

```
struct executor_arg_t
```

Member Functions

Name	Description
executor_arg_t	Constructor.

The `executor_arg_t` struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, some types have constructors with `executor_arg_t` as the first argument, immediately followed by an argument of a type that satisfies the Executor type requirements.

Requirements

Header: asio/uses_executor.hpp

Convenience header: asio.hpp

5.130.1 executor_arg_t::executor_arg_t

Constructor.

```
constexpr executor_arg_t();
```

5.131 executor_binder

A call wrapper type to bind an executor of type Executor to an object of type T.

```
template<
    typename T,
    typename Executor>
class executor_binder
```

Types

Name	Description
argument_type	The type of the function's argument.
executor_type	The type of the associated executor.

Name	Description
first_argument_type	The type of the function's first argument.
result_type	The return type if a function.
second_argument_type	The type of the function's second argument.
target_type	The type of the target object.

Member Functions

Name	Description
executor_binder	Construct an executor wrapper for the specified object. Copy constructor. Construct a copy, but specify a different executor. Construct a copy of a different executor wrapper type. Construct a copy of a different executor wrapper type, but specify a different executor. Move constructor. Move construct the target object, but specify a different executor. Move construct from a different executor wrapper type. Move construct from a different executor wrapper type, but specify a different executor.
get	Obtain a reference to the target object.
get_executor	Obtain the associated executor.
operator()	
~executor_binder	Destructor.

Requirements

Header: asio/bind_executor.hpp

Convenience header: asio.hpp

5.131.1 executor_binder::argument_type

The type of the function's argument.

```
typedef see_below argument_type;
```

The type of argument_type is based on the type T of the wrapper's target object:

- if T is a pointer to a function type accepting a single argument, argument_type is a synonym for the return type of T;
- if T is a class type with a member type argument_type, then argument_type is a synonym for T::argument_type;
- otherwise argument_type is not defined.

Requirements

Header: asio/bind_executor.hpp

Convenience header: asio.hpp

5.131.2 executor_binder::executor_binder

Construct an executor wrapper for the specified object.

```
template<
    typename U>
executor_binder(
    executor_arg_t ,
    const executor_type & e,
    U && u);
```

Copy constructor.

```
executor_binder(
    const executor_binder & other);
```

Construct a copy, but specify a different executor.

```
executor_binder(
    executor_arg_t ,
    const executor_type & e,
    const executor_binder & other);
```

Construct a copy of a different executor wrapper type.

```
template<
    typename U,
    typename OtherExecutor>
executor_binder(
    const executor_binder< U, OtherExecutor > & other);
```

Construct a copy of a different executor wrapper type, but specify a different executor.

```
template<
    typename U,
    typename OtherExecutor>
executor_binder(
    executor_arg_t ,
    const executor_type & e,
    const executor_binder< U, OtherExecutor > & other);
```

Move constructor.

```
executor_binder(
    executor_binder && other);
```

Move construct the target object, but specify a different executor.

```
executor_binder(
    executor_arg_t ,
    const executor_type & e,
    executor_binder && other);
```

Move construct from a different executor wrapper type.

```
template<
    typename U,
    typename OtherExecutor>
executor_binder(
    executor_binder< U, OtherExecutor > && other);
```

Move construct from a different executor wrapper type, but specify a different executor.

```
template<
    typename U,
    typename OtherExecutor>
executor_binder(
    executor_arg_t ,
    const executor_type & e,
    executor_binder< U, OtherExecutor > && other);
```

5.131.2.1 executor_binder::executor_binder (1 of 9 overloads)

Construct an executor wrapper for the specified object.

```
template<
    typename U>
executor_binder(
    executor_arg_t ,
    const executor_type & e,
    U && u);
```

This constructor is only valid if the type T is constructible from type U.

5.131.2.2 executor_binder::executor_binder (2 of 9 overloads)

Copy constructor.

```
executor_binder(
    const executor_binder & other);
```

5.131.2.3 executor_binder::executor_binder (3 of 9 overloads)

Construct a copy, but specify a different executor.

```
executor_binder(
    executor_arg_t ,
    const executor_type & e,
    const executor_binder & other);
```

5.131.2.4 executor_binder::executor_binder (4 of 9 overloads)

Construct a copy of a different executor wrapper type.

```
template<
    typename U,
    typename OtherExecutor>
executor_binder(
    const executor_binder< U, OtherExecutor > & other);
```

This constructor is only valid if the `Executor` type is constructible from type `OtherExecutor`, and the type `T` is constructible from type `U`.

5.131.2.5 executor_binder::executor_binder (5 of 9 overloads)

Construct a copy of a different executor wrapper type, but specify a different executor.

```
template<
    typename U,
    typename OtherExecutor>
executor_binder(
    executor_arg_t ,
    const executor_type & e,
    const executor_binder< U, OtherExecutor > & other);
```

This constructor is only valid if the type `T` is constructible from type `U`.

5.131.2.6 executor_binder::executor_binder (6 of 9 overloads)

Move constructor.

```
executor_binder(
    executor_binder && other);
```

5.131.2.7 executor_binder::executor_binder (7 of 9 overloads)

Move construct the target object, but specify a different executor.

```
executor_binder(
    executor_arg_t ,
    const executor_type & e,
    executor_binder && other);
```

5.131.2.8 executor_binder::executor_binder (8 of 9 overloads)

Move construct from a different executor wrapper type.

```
template<
    typename U,
    typename OtherExecutor>
executor_binder(
    executor_binder< U, OtherExecutor > && other);
```

5.131.2.9 executor_binder::executor_binder (9 of 9 overloads)

Move construct from a different executor wrapper type, but specify a different executor.

```
template<
    typename U,
    typename OtherExecutor>
executor_binder(
    executor_arg_t ,
    const executor_type & e,
    executor_binder< U, OtherExecutor > && other);
```

5.131.3 executor_binder::executor_type

The type of the associated executor.

```
typedef Executor executor_type;
```

Requirements

Header: asio/bind_executor.hpp

Convenience header: asio.hpp

5.131.4 executor_binder::first_argument_type

The type of the function's first argument.

```
typedef see_below first_argument_type;
```

The type of `first_argument_type` is based on the type `T` of the wrapper's target object:

- if `T` is a pointer to a function type accepting two arguments, `first_argument_type` is a synonym for the return type of `T`;
- if `T` is a class type with a member type `first_argument_type`, then `first_argument_type` is a synonym for `T::first_argument_type`;
- otherwise `first_argument_type` is not defined.

Requirements

Header: asio/bind_executor.hpp

Convenience header: asio.hpp

5.131.5 executor_binder::get

Obtain a reference to the target object.

```
target_type & get();  
  
const target_type & get() const;
```

5.131.5.1 executor_binder::get (1 of 2 overloads)

Obtain a reference to the target object.

```
target_type & get();
```

5.131.5.2 executor_binder::get (2 of 2 overloads)

Obtain a reference to the target object.

```
const target_type & get() const;
```

5.131.6 executor_binder::get_executor

Obtain the associated executor.

```
executor_type get_executor() const;
```

5.131.7 executor_binder::operator()

```
template<
    typename... Args>
auto operator()(Args && ...) ;

template<
    typename... Args>
auto operator()(Args && ...) const;
```

5.131.7.1 executor_binder::operator() (1 of 2 overloads)

```
template<
    typename... Args>
auto operator()(Args && ...);
```

5.131.7.2 executor_binder::operator() (2 of 2 overloads)

```
template<
    typename... Args>
auto operator()(Args && ...) const;
```

5.131.8 executor_binder::result_type

The return type if a function.

```
typedef see_below result_type;
```

The type of `result_type` is based on the type `T` of the wrapper's target object:

- if `T` is a pointer to function type, `result_type` is a synonym for the return type of `T`;
- if `T` is a class type with a member type `result_type`, then `result_type` is a synonym for `T::result_type`;
- otherwise `result_type` is not defined.

Requirements

Header: asio/bind_executor.hpp

Convenience header: asio.hpp

5.131.9 executor_binder::second_argument_type

The type of the function's second argument.

```
typedef see_below second_argument_type;
```

The type of `second_argument_type` is based on the type `T` of the wrapper's target object:

- if `T` is a pointer to a function type accepting two arguments, `second_argument_type` is a synonym for the return type of `T`;
- if `T` is a class type with a member type `first_argument_type`, then `second_argument_type` is a synonym for `T::second_argument_type`;
- otherwise `second_argument_type` is not defined.

Requirements

Header: asio/bind_executor.hpp

Convenience header: asio.hpp

5.131.10 executor_binder::target_type

The type of the target object.

```
typedef T target_type;
```

Requirements

Header: asio/bind_executor.hpp

Convenience header: asio.hpp

5.131.11 executor_binder::~executor_binder

Destructor.

```
~executor_binder();
```

5.132 executor_work_guard

An object of type `executor_work_guard` controls ownership of executor work within a scope.

```
template<
    typename Executor>
class executor_work_guard
```

Types

Name	Description
executor_type	The underlying executor type.

Member Functions

Name	Description
executor_work_guard	Constructs a executor_work_guard object for the specified executor. Copy constructor. Move constructor.
get_executor	Obtain the associated executor.
owns_work	Whether the executor_work_guard object owns some outstanding work.
reset	Indicate that the work is no longer outstanding.
~executor_work_guard	Destructor.

Requirements

Header: asio/executor_work_guard.hpp

Convenience header: asio.hpp

5.132.1 executor_work_guard::executor_type

The underlying executor type.

```
typedef Executor executor_type;
```

Requirements

Header: asio/executor_work_guard.hpp

Convenience header: asio.hpp

5.132.2 executor_work_guard::executor_work_guard

Constructs a executor_work_guard object for the specified executor.

```
explicit executor_work_guard(
    const executor_type & e);
```

Copy constructor.

```
executor_work_guard(
    const executor_work_guard & other);
```

Move constructor.

```
executor_work_guard(
    executor_work_guard && other);
```

5.132.2.1 `executor_work_guard::executor_work_guard (1 of 3 overloads)`

Constructs a `executor_work_guard` object for the specified executor.

```
executor_work_guard(
    const executor_type & e);
```

Stores a copy of `e` and calls `on_work_started()` on it.

5.132.2.2 `executor_work_guard::executor_work_guard (2 of 3 overloads)`

Copy constructor.

```
executor_work_guard(
    const executor_work_guard & other);
```

5.132.2.3 `executor_work_guard::executor_work_guard (3 of 3 overloads)`

Move constructor.

```
executor_work_guard(
    executor_work_guard && other);
```

5.132.3 `executor_work_guard::get_executor`

Obtain the associated executor.

```
executor_type get_executor() const;
```

5.132.4 `executor_work_guard::owns_work`

Whether the `executor_work_guard` object owns some outstanding work.

```
bool owns_work() const;
```

5.132.5 `executor_work_guard::reset`

Indicate that the work is no longer outstanding.

```
void reset();
```

5.132.6 `executor_work_guard::~executor_work_guard`

Destructor.

```
~executor_work_guard();
```

Unless the object has already been reset, or is in a moved-from state, calls `on_work_finished()` on the stored executor.

5.133 generic::basic_endpoint

Describes an endpoint for any socket type.

```
template<
    typename Protocol>
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
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/basic_endpoint.hpp`

Convenience header: `asio.hpp`

5.133.1 `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);
```

5.133.1.1 `generic::basic_endpoint::basic_endpoint (1 of 4 overloads)`

Default constructor.

```
basic_endpoint();
```

5.133.1.2 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);
```

5.133.1.3 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);
```

5.133.1.4 generic::basic_endpoint::basic_endpoint (4 of 4 overloads)

Copy constructor.

```
basic_endpoint(
    const basic_endpoint & other);
```

5.133.2 generic::basic_endpoint::capacity

Get the capacity of the endpoint in the native type.

```
std::size_t capacity() const;
```

5.133.3 generic::basic_endpoint::data

Get the underlying endpoint in the native type.

```
data_type * data();

const data_type * data() const;
```

5.133.3.1 generic::basic_endpoint::data (1 of 2 overloads)

Get the underlying endpoint in the native type.

```
data_type * data();
```

5.133.3.2 generic::basic_endpoint::data (2 of 2 overloads)

Get the underlying endpoint in the native type.

```
const data_type * data() const;
```

5.133.4 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

5.133.5 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

5.133.6 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

5.133.7 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

5.133.8 generic::basic_endpoint::operator=

Assign from another endpoint.

```
basic_endpoint & operator=(  
    const basic_endpoint & other);
```

5.133.9 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

5.133.10 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

5.133.11 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

5.133.12 generic::basic_endpoint::protocol

The protocol associated with the endpoint.

```
protocol_type protocol() const;
```

5.133.13 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

5.133.14 generic::basic_endpoint::resize

Set the underlying size of the endpoint in the native type.

```
void resize(  
    std::size_t new_size);
```

5.133.15 generic::basic_endpoint::size

Get the underlying size of the endpoint in the native type.

```
std::size_t size() const;
```

5.134 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.

Name	Description
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

5.134.1 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);
```

5.134.1.1 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);
```

5.134.1.2 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.

5.134.2 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.

Name	Description
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`

5.134.3 generic::datagram_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.134.4 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

5.134.5 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

5.134.6 generic::datagram_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.134.7 generic::datagram_protocol::socket

The generic 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.

Name	Description
enable_connection_aborted	Socket option to report aborted connections on accept.
endpoint_type	The endpoint type.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.

Name	Description
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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.

Name	Description
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.
release	Release ownership of the underlying native socket.
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
~basic_datagram_socket	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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

5.134.8 generic::datagram_protocol::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.135 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.

Name	Description
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

5.135.1 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.
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`

5.135.2 generic::raw_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.135.3 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

5.135.4 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

5.135.5 generic::raw_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.135.6 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);
```

5.135.6.1 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);
```

5.135.6.2 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.

5.135.7 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.
executor_type	The type of the executor associated with the 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.

Name	Description
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use <code>get_executor()</code>) Get the io_context associated with the object.
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code>) Get the io_context 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_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.

Name	Description
receive_from	Receive raw data with the endpoint of the sender.
release	Release ownership of the underlying native socket.
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
~basic_raw_socket	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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`

5.135.8 generic::raw_protocol::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.136 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

5.136.1 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`

5.136.2 generic::seq_packet_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.136.3 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`

5.136.4 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

5.136.5 generic::seq_packet_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.136.6 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);
```

5.136.6.1 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);
```

5.136.6.2 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.

5.136.7 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.
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.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.

Name	Description
send_low_watermark	Socket option for the send low watermark.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context associated with the object.

Name	Description
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_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_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.
release	Release ownership of the underlying native 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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
~basic_seq_packet_socket	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.

Name	Description
max_listen_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.

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`

5.136.8 generic::seq_packet_protocol::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.137 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.
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`

5.137.1 `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

5.137.2 generic::stream_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.137.3 generic::stream_protocol::iostream

The generic socket iostream type.

```
typedef basic_socket_iostream< stream_protocol > iostream;
```

Types

Name	Description
clock_type	The clock type.
duration	The duration type.
duration_type	(Deprecated: Use duration.) The duration type.
endpoint_type	The endpoint type.
protocol_type	The protocol type.
time_point	The time type.
time_type	(Deprecated: Use time_point.) The time type.

Member Functions

Name	Description
<code>basic_socket_iostream</code>	Construct a <code>basic_socket_iostream</code> without establishing a connection. Construct a <code>basic_socket_iostream</code> from the supplied socket. Move-construct a <code>basic_socket_iostream</code> from another. Establish a connection to an endpoint corresponding to a resolver query.
<code>close</code>	Close the connection.
<code>connect</code>	Establish a connection to an endpoint corresponding to a resolver query.
<code>error</code>	Get the last error associated with the stream.
<code>expires_after</code>	Set the stream's expiry time relative to now.
<code>expires_at</code>	(Deprecated: Use <code>expiry()</code> .) Get the stream's expiry time as an absolute time. Set the stream's expiry time as an absolute time.
<code>expires_from_now</code>	(Deprecated: Use <code>expiry()</code> .) Get the stream's expiry time relative to now. (Deprecated: Use <code>expires_after()</code> .) Set the stream's expiry time relative to now.
<code>expiry</code>	Get the stream's expiry time as an absolute time.
<code>operator=</code>	Move-assign a <code>basic_socket_iostream</code> from another.
<code>rdbuf</code>	Return a pointer to the underlying <code>streambuf</code> .
<code>socket</code>	Get a reference to the underlying socket.

Requirements

Header: `asio/generic/stream_protocol.hpp`

Convenience header: `asio.hpp`

5.137.4 `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`

5.137.5 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

5.137.6 generic::stream_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.137.7 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.
executor_type	The type of the executor associated with the 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.

Name	Description
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</code> associated with the object.
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</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_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.

Name	Description
receive	Receive some data on the socket. Receive some data on a connected socket.
release	Release ownership of the underlying native 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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
write_some	Write some data to the socket.
~basic_stream_socket	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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`

5.137.8 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);
```

5.137.8.1 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);
```

5.137.8.2 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.

5.137.9 generic::stream_protocol::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.138 get_associated_allocator

Helper function to obtain an object's associated allocator.

```
template<
    typename T>
associated_allocator< T >::type get_associated_allocator(
    const T & t);

template<
    typename T,
```

```
    typename Allocator>
associated_allocator< T, Allocator >::type get_associated_allocator(
    const T & t,
    const Allocator & a);
```

Requirements

Header: asio/associated_allocator.hpp

Convenience header: asio.hpp

5.138.1 get_associated_allocator (1 of 2 overloads)

Helper function to obtain an object's associated allocator.

```
template<
    typename T>
associated_allocator< T >::type get_associated_allocator(
    const T & t);
```

Return Value

```
associated_allocator<T>::get(t)
```

5.138.2 get_associated_allocator (2 of 2 overloads)

Helper function to obtain an object's associated allocator.

```
template<
    typename T,
    typename Allocator>
associated_allocator< T, Allocator >::type get_associated_allocator(
    const T & t,
    const Allocator & a);
```

Return Value

```
associated_allocator<T, Allocator>::get(t, a)
```

5.139 get_associated_executor

Helper function to obtain an object's associated executor.

```
template<
    typename T>
associated_executor< T >::type get_associated_executor(
    const T & t);

template<
    typename T,
    typename Executor>
associated_executor< T, Executor >::type get_associated_executor(
    const T & t,
```

```

const Executor & ex,
typename enable_if< is_executor< Executor >::value >::type * = 0);

template<
    typename T,
    typename ExecutionContext>
associated_executor< T, typename ExecutionContext::executor_type >::type ←
    get_associated_executor(
        const T & t,
        ExecutionContext & ctx,
        typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::type * = 0);

```

Requirements

Header: asio/associated_executor.hpp

Convenience header: asio.hpp

5.139.1 get_associated_executor (1 of 3 overloads)

Helper function to obtain an object's associated executor.

```

template<
    typename T>
associated_executor< T >::type get_associated_executor(
    const T & t);

```

Return Value

associated_executor<T>::get(t)

5.139.2 get_associated_executor (2 of 3 overloads)

Helper function to obtain an object's associated executor.

```

template<
    typename T,
    typename Executor>
associated_executor< T, Executor >::type get_associated_executor(
    const T & t,
    const Executor & ex,
    typename enable_if< is_executor< Executor >::value >::type * = 0);

```

Return Value

associated_executor<T, Executor>::get(t, ex)

5.139.3 get_associated_executor (3 of 3 overloads)

Helper function to obtain an object's associated executor.

```
template<
    typename T,
    typename ExecutionContext>
associated_executor< T, typename ExecutionContext::executor_type >::type ←
    get_associated_executor(
        const T & t,
        ExecutionContext & ctx,
        typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::type * = 0);
```

Return Value

```
associated_executor<T, typename ExecutionContext::executor_type>::get(t, ctx.get_executor())
```

5.140 handler_type

(Deprecated: Use two-parameter version of [async_result](#).) Default handler type traits provided for all completion token types.

```
template<
    typename CompletionToken,
    typename Signature,
    typename = void>
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 completion token types.

Requirements

Header: `asio/handler_type.hpp`

Convenience header: `asio.hpp`

5.140.1 handler_type::type

The handler type for the specific signature.

```
typedef conditional< is_same< CompletionToken, typename decay< CompletionToken >::type >::value ←
    , decay< CompletionToken >, handler_type< typename decay< CompletionToken >::type, ←
    Signature > >::type::type type;
```

Requirements

Header: asio/handler_type.hpp

Convenience header: asio.hpp

5.141 has_service

```
template<
    typename Service>
bool has_service(
    execution_context & e);
```

This function is used to determine whether the `execution_context` contains a service object corresponding to the given service type.

Parameters

`e` The `execution_context` object that owns the service.

Return Value

A boolean indicating whether the `execution_context` contains the service.

Requirements

Header: asio/impl/execution_context.hpp

Convenience header: asio.hpp

5.142 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>executor_type</code>	The type of the executor associated with the object.
<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_waitable_timer</code>	Constructor. Constructor to set a particular expiry time as an absolute time. Constructor to set a particular expiry time relative to now. Move-construct a basic_waitable_timer from another.
<code>cancel</code>	Cancel any asynchronous operations that are waiting on the timer. (Deprecated: Use non-error_code overload.) Cancel any asynchronous operations that are waiting on the timer.
<code>cancel_one</code>	Cancels one asynchronous operation that is waiting on the timer. (Deprecated: Use non-error_code overload.) Cancels one asynchronous operation that is waiting on the timer.
<code>expires_after</code>	Set the timer's expiry time relative to now.
<code>expires_at</code>	(Deprecated: Use expiry().) Get the timer's expiry time as an absolute time. Set the timer's expiry time as an absolute time. (Deprecated: Use non-error_code overload.) Set the timer's expiry time as an absolute time.
<code>expires_from_now</code>	(Deprecated: Use expiry().) Get the timer's expiry time relative to now. (Deprecated: Use expires_after().) Set the timer's expiry time relative to now.
<code>expiry</code>	Get the timer's expiry time as an absolute time.
<code>get_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use get_executor().) Get the io_context associated with the object.
<code>get_io_service</code>	(Deprecated: Use get_executor().) Get the io_context associated with the object.
<code>operator=</code>	Move-assign a basic_waitable_timer from another.
<code>wait</code>	Perform a blocking wait on the timer.
<code>~basic_waitable_timer</code>	Destroys the timer.

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_context);  
  
// Set an expiry time relative to now.  
timer.expires_after(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_context,  
    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_after(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_after()` 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: `asio.hpp`

5.143 invalid_service_owner

Exception thrown when trying to add a service object to an `execution_context` where the service has a different owner.

```
class invalid_service_owner
```

Member Functions

Name	Description
<code>invalid_service_owner</code>	

Requirements

Header: `asio/execution_context.hpp`

Convenience header: `asio.hpp`

5.143.1 invalid_service_owner::invalid_service_owner

```
invalid_service_owner();
```

5.144 io_context

Provides core I/O functionality.

```
class io_context :  
    public execution_context
```

Types

Name	Description
executor_type	Executor used to submit functions to an io_context.
service	Base class for all io_context services.
strand	Provides serialised handler execution.
work	(Deprecated: Use executor_work_guard.) Class to inform the io_context when it has work to do.
count_type	The type used to count the number of handlers executed by the context.
fork_event	Fork-related event notifications.

Member Functions

Name	Description
dispatch	(Deprecated: Use asio::dispatch().) Request the io_context to invoke the given handler.
get_executor	Obtains the executor associated with the io_context.
io_context	Constructor.
notify_fork	Notify the execution_context of a fork-related event.
poll	Run the io_context object's event processing loop to execute ready handlers. (Deprecated: Use non-error_code overload.) Run the io_context object's event processing loop to execute ready handlers.
poll_one	Run the io_context object's event processing loop to execute one ready handler. (Deprecated: Use non-error_code overload.) Run the io_context object's event processing loop to execute one ready handler.
post	(Deprecated: Use asio::post().) Request the io_context to invoke the given handler and return immediately.
reset	(Deprecated: Use restart().) Reset the io_context in preparation for a subsequent run() invocation.

Name	Description
restart	Restart the io_context in preparation for a subsequent run() invocation.
run	Run the io_context object's event processing loop. (Deprecated: Use non-error_code overload.) Run the io_context object's event processing loop.
run_for	Run the io_context object's event processing loop for a specified duration.
run_one	Run the io_context object's event processing loop to execute at most one handler. (Deprecated: Use non-error_code overload.) Run the io_context object's event processing loop to execute at most one handler.
run_one_for	Run the io_context object's event processing loop for a specified duration to execute at most one handler.
run_one_until	Run the io_context object's event processing loop until a specified time to execute at most one handler.
run_until	Run the io_context object's event processing loop until a specified time.
stop	Stop the io_context object's event processing loop.
stopped	Determine whether the io_context object has been stopped.
wrap	(Deprecated: Use asio::bind_executor().) Create a new handler that automatically dispatches the wrapped handler on the io_context.
~io_context	Destructor.

Protected Member Functions

Name	Description
destroy	Destroys all services in the context.
shutdown	Shuts down all services in the context.

Friends

Name	Description
add_service	(Deprecated: Use make_service().) Add a service object to the execution_context.

Name	Description
has_service	Determine if an execution_context contains a specified service type.
make_service	Creates a service object and adds it to the execution_context.
use_service	Obtain the service object corresponding to the given type.

The `io_context` 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_context` 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 `restart()` and `notify_fork()` functions. Calling `restart()` while there are unfinished `run()`, `run_one()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` calls results in undefined behaviour. The `notify_fork()` function should not be called while any `io_context` function, or any function on an I/O object that is associated with the `io_context`, is being called in another thread.

Synchronous and asynchronous operations

Synchronous operations on I/O objects implicitly run the `io_context` object for an individual operation. The `io_context` functions `run()`, `run_one()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` must be called for the `io_context` 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()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` for the `io_context`.

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()`, `run_for()`, `run_until()`, `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()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` call may be restarted *without* the need for an intervening call to `restart()`. This allows the thread to rejoin the `io_context` object's thread pool without impacting any other threads in the pool.

For example:

```
asio::io_context io_context;
...
for (;;)
{
    try
    {
        io_context.run();
    }
}
```

```

        break; // run() exited normally
    }
    catch (my_exception& e)
    {
        // Deal with exception as appropriate.
    }
}

```

Submitting arbitrary tasks to the io_context

To submit functions to the `io_context`, use the `dispatch`, `post` or `defer` free functions.

For example:

```

void my_task()
{
    ...
}

asio::io_context io_context;

// Submit a function to the io_context.
asio::post(io_context, my_task);

// Submit a lambda object to the io_context.
asio::post(io_context,
    []()
    {
        ...
    });
}

// Run the io_context until it runs out of work.
io_context.run();

```

Stopping the io_context from running out of work

Some applications may need to prevent an `io_context` object's `run()` call from returning when there is no more work to do. For example, the `io_context` 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 `asio::executor_work_guard<io_context::executor_type>`:

```

asio::io_context io_context;
asio::executor_work_guard<asio::io_context::executor_type>
    = asio::make_work_guard(io_context);
...

```

To effect a shutdown, the application will then need to call the `io_context` object's `stop()` member function. This will cause the `io_context` `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 reset.

```

asio::io_context io_context;
asio::executor_work_guard<asio::io_context::executor_type>
    = asio::make_work_guard(io_context);
...
work.reset(); // Allow run() to exit.

```

Requirements

Header: asio/io_context.hpp

Convenience header: asio.hpp

5.144.1 io_context::add_service

Inherited from execution_context.

(Deprecated: Use `make_service()`.) Add a service object to the `execution_context`.

```
template<
    typename Service>
friend void add_service(
    execution_context & e,
    Service * svc);
```

This function is used to add a service to the `execution_context`.

Parameters

e The `execution_context` object that owns the service.

svc The service object. On success, ownership of the service object is transferred to the `execution_context`. When the `execution_context` object is destroyed, it will destroy the service object by performing:

```
delete static_cast<execution_context::service*>(svc)
```

Exceptions

asio::service_already_exists Thrown if a service of the given type is already present in the `execution_context`.

asio::invalid_service_owner Thrown if the service's owning `execution_context` is not the `execution_context` object specified by the `e` parameter.

Requirements

Header: asio/io_context.hpp

Convenience header: asio.hpp

5.144.2 io_context::count_type

The type used to count the number of handlers executed by the context.

```
typedef std::size_t count_type;
```

Requirements

Header: asio/io_context.hpp

Convenience header: asio.hpp

5.144.3 io_context::destroy

Inherited from execution_context.

Destroys all services in the context.

```
void destroy();
```

This function is implemented as follows:

- For each service object `svc` in the `execution_context` set, in reverse order * of the beginning of service object lifetime, performs `delete static_cast<execution_context::service*>(svc)`.

5.144.4 io_context::dispatch

(Deprecated: Use `dispatch`.) Request the `io_context` to invoke the given handler.

```
template<
    typename CompletionHandler>
DEDUCED dispatch(
    CompletionHandler && handler);
```

This function is used to ask the `io_context` to execute the given handler.

The `io_context` 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. The handler may be executed inside this function if the guarantee can be met.

Parameters

handler The handler to be called. The `io_context` 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.

5.144.5 io_context::fork_event

Inherited from execution_context.

Fork-related event notifications.

```
enum fork_event
```

Values

fork_prepare Notify the context that the process is about to fork.

fork_parent Notify the context that the process has forked and is the parent.

fork_child Notify the context that the process has forked and is the child.

5.144.6 io_context::get_executor

Obtains the executor associated with the [io_context](#).

```
executor_type get_executor();
```

5.144.7 io_context::has_service

Inherited from execution_context.

Determine if an [execution_context](#) contains a specified service type.

```
template<
    typename Service>
friend bool has_service(
    execution_context & e);
```

This function is used to determine whether the [execution_context](#) contains a service object corresponding to the given service type.

Parameters

e The [execution_context](#) object that owns the service.

Return Value

A boolean indicating whether the [execution_context](#) contains the service.

Requirements

Header: asio/io_context.hpp

Convenience header: asio.hpp

5.144.8 io_context::io_context

Constructor.

```
io_context();
explicit io_context(
    int concurrency_hint);
```

5.144.8.1 `io_context::io_context (1 of 2 overloads)`

Constructor.

```
io_context();
```

5.144.8.2 `io_context::io_context (2 of 2 overloads)`

Constructor.

```
io_context(  
    int 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.

5.144.9 `io_context::make_service`

Inherited from execution_context.

Creates a service object and adds it to the [execution_context](#).

```
template<  
    typename Service,  
    typename... Args>  
friend Service & make_service(  
    execution_context & e,  
    Args &&... args);
```

This function is used to add a service to the [execution_context](#).

Parameters

e The [execution_context](#) object that owns the service.

args Zero or more arguments to be passed to the service constructor.

Exceptions

asio::service_already_exists Thrown if a service of the given type is already present in the [execution_context](#).

Requirements

Header: `asio/io_context.hpp`

Convenience header: `asio.hpp`

5.144.10 io_context::notify_fork

Inherited from execution_context.

Notify the execution_context of a fork-related event.

```
void notify_fork(  
    fork_event event);
```

This function is used to inform the execution_context that the process is about to fork, or has just forked. This allows the execution_context, 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 execution_context function, or any function associated with the execution_context's derived class, 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 execution_context or its derived class.

Parameters

event A fork-related event.

Exceptions

asio::system_error Thrown on failure. If the notification fails the execution_context object should no longer be used and should be destroyed.

Example

The following code illustrates how to incorporate the notify_fork() function:

```
my_execution_context.notify_fork(execution_context::fork_prepare);  
if (fork() == 0)  
{  
    // This is the child process.  
    my_execution_context.notify_fork(execution_context::fork_child);  
}  
else  
{  
    // This is the parent process.  
    my_execution_context.notify_fork(execution_context::fork_parent);  
}
```

Remarks

For each service object svc in the execution_context set, performs svc->notify_fork();. 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.

5.144.11 io_context::poll

Run the io_context object's event processing loop to execute ready handlers.

```
count_type poll();
```

(Deprecated: Use non-error_code overload.) Run the io_context object's event processing loop to execute ready handlers.

```
count_type poll(  
    asio::error_code & ec);
```

5.144.11.1 `io_context::poll` (1 of 2 overloads)

Run the `io_context` object's event processing loop to execute ready handlers.

```
count_type poll();
```

The `poll()` function runs handlers that are ready to run, without blocking, until the `io_context` has been stopped or there are no more ready handlers.

Return Value

The number of handlers that were executed.

5.144.11.2 `io_context::poll` (2 of 2 overloads)

(Deprecated: Use non-error_code overload.) Run the `io_context` object's event processing loop to execute ready handlers.

```
count_type poll(  
    asio::error_code & ec);
```

The `poll()` function runs handlers that are ready to run, without blocking, until the `io_context` 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.

5.144.12 `io_context::poll_one`

Run the `io_context` object's event processing loop to execute one ready handler.

```
count_type poll_one();
```

(Deprecated: Use non-error_code overload.) Run the `io_context` object's event processing loop to execute one ready handler.

```
count_type poll_one(  
    asio::error_code & ec);
```

5.144.12.1 `io_context::poll_one` (1 of 2 overloads)

Run the `io_context` object's event processing loop to execute one ready handler.

```
count_type 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.

5.144.12.2 `io_context::poll_one` (2 of 2 overloads)

(Deprecated: Use non-error_code overload.) Run the `io_context` object's event processing loop to execute one ready handler.

```
count_type 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.

5.144.13 `io_context::post`

(Deprecated: Use `post`.) Request the `io_context` to invoke the given handler and return immediately.

```
template<
    typename CompletionHandler>
DEDUCED post(
    CompletionHandler && handler);
```

This function is used to ask the `io_context` to execute the given handler, but without allowing the `io_context` to call the handler from inside this function.

The `io_context` 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_context` 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.

5.144.14 `io_context::reset`

(Deprecated: Use `restart()`.) Reset the `io_context` 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_context` being stopped or running out of work. After a call to `reset()`, the `io_context` 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.

5.144.15 `io_context::restart`

Restart the `io_context` in preparation for a subsequent `run()` invocation.

```
void restart();
```

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_context` being stopped or running out of work. After a call to `restart()`, the `io_context` 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.

5.144.16 `io_context::run`

Run the `io_context` object's event processing loop.

```
count_type run();
```

(Deprecated: Use non-error_code overload.) Run the `io_context` object's event processing loop.

```
count_type run(  
    asio::error_code & ec);
```

5.144.16.1 `io_context::run (1 of 2 overloads)`

Run the `io_context` object's event processing loop.

```
count_type run();
```

The `run()` function blocks until all work has finished and there are no more handlers to be dispatched, or until the `io_context` has been stopped.

Multiple threads may call the `run()` function to set up a pool of threads from which the `io_context` may execute handlers. All threads that are waiting in the pool are equivalent and the `io_context` may choose any one of them to invoke a handler.

A normal exit from the `run()` function implies that the `io_context` 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 `restart()`.

Return Value

The number of handlers that were executed.

Remarks

Calling the `run()` function from a thread that is currently calling one of `run()`, `run_one()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` on the same `io_context` object may introduce the potential for deadlock. It is the caller's responsibility to avoid this.

The `poll()` function may also be used to dispatch ready handlers, but without blocking.

5.144.16.2 `io_context::run` (2 of 2 overloads)

(Deprecated: Use non-error_code overload.) Run the `io_context` object's event processing loop.

```
count_type 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_context` has been stopped.

Multiple threads may call the `run()` function to set up a pool of threads from which the `io_context` may execute handlers. All threads that are waiting in the pool are equivalent and the `io_context` may choose any one of them to invoke a handler.

A normal exit from the `run()` function implies that the `io_context` 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 `restart()`.

Parameters

ec Set to indicate what error occurred, if any.

Return Value

The number of handlers that were executed.

Remarks

Calling the `run()` function from a thread that is currently calling one of `run()`, `run_one()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` on the same `io_context` object may introduce the potential for deadlock. It is the caller's responsibility to avoid this.

The `poll()` function may also be used to dispatch ready handlers, but without blocking.

5.144.17 `io_context::run_for`

Run the `io_context` object's event processing loop for a specified duration.

```
template<
    typename Rep,
    typename Period>
std::size_t run_for(
    const chrono::duration<Rep, Period> & rel_time);
```

The `run_for()` function blocks until all work has finished and there are no more handlers to be dispatched, until the `io_context` has been stopped, or until the specified duration has elapsed.

Parameters

rel_time The duration for which the call may block.

Return Value

The number of handlers that were executed.

5.144.18 `io_context::run_one`

Run the `io_context` object's event processing loop to execute at most one handler.

```
count_type run_one();
```

(Deprecated: Use non-error_code overload.) Run the `io_context` object's event processing loop to execute at most one handler.

```
count_type run_one(  
    asio::error_code & ec);
```

5.144.18.1 `io_context::run_one` (1 of 2 overloads)

Run the `io_context` object's event processing loop to execute at most one handler.

```
count_type run_one();
```

The `run_one()` function blocks until one handler has been dispatched, or until the `io_context` has been stopped.

Return Value

The number of handlers that were executed. A zero return value implies that the `io_context` object is stopped (the `stopped()` function returns `true`). Subsequent calls to `run()`, `run_one()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` will return immediately unless there is a prior call to `restart()`.

Remarks

Calling the `run_one()` function from a thread that is currently calling one of `run()`, `run_one()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` on the same `io_context` object may introduce the potential for deadlock. It is the caller's responsibility to avoid this.

5.144.18.2 `io_context::run_one` (2 of 2 overloads)

(Deprecated: Use non-error_code overload.) Run the `io_context` object's event processing loop to execute at most one handler.

```
count_type run_one(  
    asio::error_code & ec);
```

The `run_one()` function blocks until one handler has been dispatched, or until the `io_context` has been stopped.

Return Value

The number of handlers that were executed. A zero return value implies that the `io_context` 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 `restart()`.

The number of handlers that were executed.

Remarks

Calling the `run_one()` function from a thread that is currently calling one of `run()`, `run_one()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` on the same `io_context` object may introduce the potential for deadlock. It is the caller's responsibility to avoid this.

5.144.19 `io_context::run_one_for`

Run the `io_context` object's event processing loop for a specified duration to execute at most one handler.

```
template<
    typename Rep,
    typename Period>
std::size_t run_one_for(
    const chrono::duration< Rep, Period > & rel_time);
```

The `run_one_for()` function blocks until one handler has been dispatched, until the `io_context` has been stopped, or until the specified duration has elapsed.

Parameters

rel_time The duration for which the call may block.

Return Value

The number of handlers that were executed.

5.144.20 `io_context::run_one_until`

Run the `io_context` object's event processing loop until a specified time to execute at most one handler.

```
template<
    typename Clock,
    typename Duration>
std::size_t run_one_until(
    const chrono::time_point< Clock, Duration > & abs_time);
```

The `run_one_until()` function blocks until one handler has been dispatched, until the `io_context` has been stopped, or until the specified time has been reached.

Parameters

abs_time The time point until which the call may block.

Return Value

The number of handlers that were executed.

5.144.21 `io_context::run_until`

Run the `io_context` object's event processing loop until a specified time.

```
template<
    typename Clock,
    typename Duration>
std::size_t run_until(
    const chrono::time_point<Clock, Duration> & abs_time);
```

The `run_until()` function blocks until all work has finished and there are no more handlers to be dispatched, until the `io_context` has been stopped, or until the specified time has been reached.

Parameters

`abs_time` The time point until which the call may block.

Return Value

The number of handlers that were executed.

5.144.22 `io_context::shutdown`

Inherited from execution_context.

Shuts down all services in the context.

```
void shutdown();
```

This function is implemented as follows:

- For each service object `svc` in the `execution_context` set, in reverse order of the beginning of service object lifetime, performs `svc->shutdown()`.

5.144.23 `io_context::stop`

Stop the `io_context` object's event processing loop.

```
void stop();
```

This function does not block, but instead simply signals the `io_context` 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 `restart()` is called.

5.144.24 `io_context::stopped`

Determine whether the `io_context` object has been stopped.

```
bool stopped() const;
```

This function is used to determine whether an `io_context` object has been stopped, either through an explicit call to `stop()`, or due to running out of work. When an `io_context` 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_context` object is stopped, otherwise false.

5.144.25 `io_context::use_service`

```
template<
    typename Service>
friend Service & use_service(
    io_context & ioc);
```

Obtain the service object corresponding to the given type.

```
template<
    typename Service>
friend Service & use_service(
    execution_context & e);
```

5.144.25.1 `io_context::use_service` (1 of 2 overloads)

```
template<
    typename Service>
friend Service & use_service(
    io_context & ioc);
```

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_context` will create a new instance of the service.

Parameters

`ioc` The `io_context` 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.

Remarks

This overload is preserved for backwards compatibility with services that inherit from `io_context::service`.

Requirements

Header: `asio/io_context.hpp`

Convenience header: `asio.hpp`

5.144.25.2 `io_context::use_service` (2 of 2 overloads)

Inherited from `execution_context`.

Obtain the service object corresponding to the given type.

```
template<
    typename Service>
friend Service & use_service(
    execution_context & e);
```

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 `execution_context` will create a new instance of the service.

Parameters

e The `execution_context` 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_context.hpp`

Convenience header: `asio.hpp`

5.144.26 `io_context::wrap`

(Deprecated: Use `bind_executor`.) Create a new handler that automatically dispatches the wrapped handler on the `io_context`.

```
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_context` object's dispatch function.

Parameters

handler The handler to be wrapped. The `io_context` 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_context` 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_context.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_context.dispatch(boost::bind(f, a1, ... an));
```

5.144.27 `io_context::~io_context`

Destructor.

```
~io_context();
```

On destruction, the `io_context` performs the following sequence of operations:

- For each service object `svc` in the `io_context` set, in reverse order of the beginning of service object lifetime, performs `svc->shutdown()`.
- Uninvoked handler objects that were scheduled for deferred invocation on the `io_context`, or any associated strand, are destroyed.
- For each service object `svc` in the `io_context` set, in reverse order of the beginning of service object lifetime, performs `delete static_cast<io_context::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_context` function `stop()` is called to terminate any `run()` calls as soon as possible. The `io_context` destructor defined above destroys all handlers, causing all `shared_ptr` references to all connection objects to be destroyed.

5.145 `io_context::executor_type`

Executor used to submit functions to an `io_context`.

```
class executor_type
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/asio.hpp

Convenience header: asio.hpp

5.145.1 io_context::executor_type::context

Obtain the underlying execution context.

```
io_context & context() const;
```

5.145.2 io_context::executor_type::defer

Request the **io_context** to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void defer(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the **io_context** to execute the given function object. The function object will never be executed inside `defer()`. Instead, it will be scheduled to run on the **io_context**.

If the current thread belongs to the **io_context**, `defer()` will delay scheduling the function object until the current thread returns control to the pool.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.145.3 io_context::executor_type::dispatch

Request the `io_context` to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void dispatch(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the `io_context` to execute the given function object. If the current thread is running the `io_context`, `dispatch()` executes the function before returning. Otherwise, the function will be scheduled to run on the `io_context`.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.145.4 io_context::executor_type::on_work_finished

Inform the `io_context` that some work is no longer outstanding.

```
void on_work_finished() const;
```

This function is used to inform the `io_context` that some work has finished. Once the count of unfinished work reaches zero, the `io_context` is stopped and the `run()` and `run_one()` functions may exit.

5.145.5 io_context::executor_type::on_work_started

Inform the `io_context` that it has some outstanding work to do.

```
void on_work_started() const;
```

This function is used to inform the `io_context` that some work has begun. This ensures that the `io_context`'s `run()` and `run_one()` functions do not exit while the work is underway.

5.145.6 `io_context::executor_type::operator!=`

Compare two executors for inequality.

```
friend bool operator!=(
    const executor_type & a,
    const executor_type & b);
```

Two executors are equal if they refer to the same underlying `io_context`.

Requirements

Header: `asio/io_context.hpp`

Convenience header: `asio.hpp`

5.145.7 `io_context::executor_type::operator==`

Compare two executors for equality.

```
friend bool operator==((
    const executor_type & a,
    const executor_type & b);
```

Two executors are equal if they refer to the same underlying `io_context`.

Requirements

Header: `asio/io_context.hpp`

Convenience header: `asio.hpp`

5.145.8 `io_context::executor_type::post`

Request the `io_context` to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void post(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the `io_context` to execute the given function object. The function object will never be executed inside `post()`. Instead, it will be scheduled to run on the `io_context`.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.145.9 `io_context::executor_type::running_in_this_thread`

Determine whether the `io_context` is running in the current thread.

```
bool running_in_this_thread() const;
```

Return Value

`true` if the current thread is running the `io_context`. Otherwise returns `false`.

5.146 `io_context::service`

Base class for all `io_context` services.

```
class service
```

Member Functions

Name	Description
<code>get_io_context</code>	Get the <code>io_context</code> object that owns the service.
<code>get_io_service</code>	Get the <code>io_context</code> object that owns the service.

Protected Member Functions

Name	Description
<code>service</code>	Constructor.
<code>~service</code>	Destructor.

Requirements

Header: `asio/io_context.hpp`

Convenience header: `asio.hpp`

5.146.1 `io_context::service::get_io_context`

Get the `io_context` object that owns the service.

```
asio::io_context & get_io_context();
```

5.146.2 `io_context::service::get_io_service`

Get the `io_context` object that owns the service.

```
asio::io_context & get_io_service();
```

5.146.3 io_context::service::service

Constructor.

```
service(  
    asio::io_context & owner);
```

Parameters

owner The `io_context` object that owns the service.

5.146.4 io_context::service::~service

Destructor.

```
virtual ~service();
```

5.147 io_context::strand

Provides serialised handler execution.

```
class strand
```

Member Functions

Name	Description
<code>context</code>	Obtain the underlying execution context.
<code>defer</code>	Request the strand to invoke the given function object.
<code>dispatch</code>	Request the strand to invoke the given function object. (Deprecated: Use <code>asio::dispatch()</code>) Request the strand to invoke the given handler.
<code>get_io_context</code>	(Deprecated: Use <code>context()</code>) Get the <code>io_context</code> associated with the strand.
<code>get_io_service</code>	(Deprecated: Use <code>context()</code>) Get the <code>io_context</code> associated with the strand.
<code>on_work_finished</code>	Inform the strand that some work is no longer outstanding.
<code>on_work_started</code>	Inform the strand that it has some outstanding work to do.
<code>post</code>	Request the strand to invoke the given function object. (Deprecated: Use <code>asio::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.

Name	Description
wrap	(Deprecated: Use <code>asio::bind_executor()</code> .) Create a new handler that automatically dispatches the wrapped handler on the strand.
<code>~strand</code>	Destructor.

Friends

Name	Description
<code>operator!=</code>	Compare two strands for inequality.
<code>operator==</code>	Compare two strands for equality.

The `io_context::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/io_context_strand.hpp`

Convenience header: `asio.hpp`

5.147.1 `io_context::strand::context`

Obtain the underlying execution context.

```
asio::io_context & context() const;
```

5.147.2 `io_context::strand::defer`

Request the strand to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void defer(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the executor to execute the given function object. The function object will never be executed inside this function. Instead, it will be scheduled to run by the underlying `io_context`.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.147.3 `io_context::strand::dispatch`

Request the strand to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void dispatch(
    Function && f,
    const Allocator & a) const;
```

(Deprecated: Use [dispatch](#).) Request the strand to invoke the given handler.

```
template<
    typename CompletionHandler>
DEDUCED dispatch(
    CompletionHandler && handler);
```

5.147.3.1 io_context::strand::dispatch (1 of 2 overloads)

Request the strand to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void dispatch(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the strand to execute the given function object on its underlying [io_context](#). The function object will be executed inside this function if the strand is not otherwise busy and if the underlying [io_context](#)'s executor's `dispatch()` function is also able to execute the function before returning.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.147.3.2 io_context::strand::dispatch (2 of 2 overloads)

(Deprecated: Use [dispatch](#).) Request the strand to invoke the given handler.

```
template<
    typename CompletionHandler>
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_context](#). The [io_context](#) guarantees that the handler will only be called in a thread in which the [io_context](#)'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();
```

5.147.4 `io_context::strand::get_io_context`

(Deprecated: Use `context()`.) Get the `io_context` associated with the strand.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the strand uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the strand will use to dispatch handlers. Ownership is not transferred to the caller.

5.147.5 `io_context::strand::get_io_service`

(Deprecated: Use `context()`.) Get the `io_context` associated with the strand.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the strand uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the strand will use to dispatch handlers. Ownership is not transferred to the caller.

5.147.6 `io_context::strand::on_work_finished`

Inform the strand that some work is no longer outstanding.

```
void on_work_finished() const;
```

The strand delegates this call to its underlying `io_context`.

5.147.7 `io_context::strand::on_work_started`

Inform the strand that it has some outstanding work to do.

```
void on_work_started() const;
```

The strand delegates this call to its underlying `io_context`.

5.147.8 `io_context::strand::operator!=`

Compare two strands for inequality.

```
friend bool operator!=(
    const strand & a,
    const strand & b);
```

Two strands are equal if they refer to the same ordered, non-concurrent state.

Requirements

Header: `asio/io_context_strand.hpp`

Convenience header: `asio.hpp`

5.147.9 `io_context::strand::operator==`

Compare two strands for equality.

```
friend bool operator==(  
    const strand & a,  
    const strand & b);
```

Two strands are equal if they refer to the same ordered, non-concurrent state.

Requirements

Header: asio/io_context_strand.hpp

Convenience header: asio.hpp

5.147.10 `io_context::strand::post`

Request the strand to invoke the given function object.

```
template<  
    typename Function,  
    typename Allocator>  
void post(  
    Function && f,  
    const Allocator & a) const;
```

(Deprecated: Use `post()`.) Request the strand to invoke the given handler and return immediately.

```
template<  
    typename CompletionHandler>  
DEDUCED post(  
    CompletionHandler && handler);
```

5.147.10.1 `io_context::strand::post (1 of 2 overloads)`

Request the strand to invoke the given function object.

```
template<  
    typename Function,  
    typename Allocator>  
void post(  
    Function && f,  
    const Allocator & a) const;
```

This function is used to ask the executor to execute the given function object. The function object will never be executed inside this function. Instead, it will be scheduled to run by the underlying `io_context`.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.147.10.2 `io_context::strand::post` (2 of 2 overloads)

(Deprecated: Use `post`.) Request the strand to invoke the given handler and return immediately.

```
template<
    typename CompletionHandler>
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_context`. The `io_context` guarantees that the handler will only be called in a thread in which the `io_context`'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();
```

5.147.11 `io_context::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.

5.147.12 `io_context::strand::strand`

Constructor.

```
strand(
    asio::io_context & io_context);
```

Constructs the strand.

Parameters

io_context The `io_context` object that the strand will use to dispatch handlers that are ready to be run.

5.147.13 `io_context::strand::wrap`

(Deprecated: Use `bind_executor`.) 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));
```

5.147.14 io_context::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.

5.148 io_context::work

(Deprecated: Use [executor_work_guard](#).) Class to inform the [io_context](#) when it has work to do.

```
class work
```

Member Functions

Name	Description
get_io_context	Get the io_context associated with the work.
get_io_service	(Deprecated: Use <code>get_io_context()</code>) Get the io_context associated with the work.
work	Constructor notifies the io_context that work is starting. Copy constructor notifies the io_context that work is starting.
~work	Destructor notifies the io_context that the work is complete.

The work class is used to inform the `io_context` when work starts and finishes. This ensures that the `io_context` 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_context.hpp`

Convenience header: `asio.hpp`

5.148.1 `io_context::work::get_io_context`

Get the `io_context` associated with the work.

```
asio::io_context & get_io_context();
```

5.148.2 `io_context::work::get_io_service`

(Deprecated: Use `get_io_context()`) Get the `io_context` associated with the work.

```
asio::io_context & get_io_service();
```

5.148.3 `io_context::work::work`

Constructor notifies the `io_context` that work is starting.

```
explicit work(
    asio::io_context & io_context);
```

Copy constructor notifies the `io_context` that work is starting.

```
work(
    const work & other);
```

5.148.3.1 `io_context::work::work (1 of 2 overloads)`

Constructor notifies the `io_context` that work is starting.

```
work(
    asio::io_context & io_context);
```

The constructor is used to inform the `io_context` that some work has begun. This ensures that the `io_context` object's `run()` function will not exit while the work is underway.

5.148.3.2 `io_context::work::work (2 of 2 overloads)`

Copy constructor notifies the `io_context` that work is starting.

```
work(
    const work & other);
```

The constructor is used to inform the `io_context` that some work has begun. This ensures that the `io_context` object's `run()` function will not exit while the work is underway.

5.148.4 `io_context::work::~work`

Destructor notifies the `io_context` that the work is complete.

```
~work();
```

The destructor is used to inform the `io_context` that some work has finished. Once the count of unfinished work reaches zero, the `io_context` object's `run()` function is permitted to exit.

5.149 `io_service`

Typedef for backwards compatibility.

```
typedef io_context io_service;
```

Types

Name	Description
<code>executor_type</code>	Executor used to submit functions to an <code>io_context</code> .
<code>service</code>	Base class for all <code>io_context</code> services.
<code>strand</code>	Provides serialised handler execution.
<code>work</code>	(Deprecated: Use <code>executor_work_guard</code> .) Class to inform the <code>io_context</code> when it has work to do.
<code>count_type</code>	The type used to count the number of handlers executed by the context.
<code>fork_event</code>	Fork-related event notifications.

Member Functions

Name	Description
<code>dispatch</code>	(Deprecated: Use <code>asio::dispatch()</code> .) Request the <code>io_context</code> to invoke the given handler.
<code>get_executor</code>	Obtains the executor associated with the <code>io_context</code> .
<code>io_context</code>	Constructor.
<code>notify_fork</code>	Notify the <code>execution_context</code> of a fork-related event.
<code>poll</code>	Run the <code>io_context</code> object's event processing loop to execute ready handlers. (Deprecated: Use non-error_code overload.) Run the <code>io_context</code> object's event processing loop to execute ready handlers.

Name	Description
poll_one	Run the io_context object's event processing loop to execute one ready handler. (Deprecated: Use non-error_code overload.) Run the io_context object's event processing loop to execute one ready handler.
post	(Deprecated: Use asio::post().) Request the io_context to invoke the given handler and return immediately.
reset	(Deprecated: Use restart().) Reset the io_context in preparation for a subsequent run() invocation.
restart	Restart the io_context in preparation for a subsequent run() invocation.
run	Run the io_context object's event processing loop. (Deprecated: Use non-error_code overload.) Run the io_context object's event processing loop.
run_for	Run the io_context object's event processing loop for a specified duration.
run_one	Run the io_context object's event processing loop to execute at most one handler. (Deprecated: Use non-error_code overload.) Run the io_context object's event processing loop to execute at most one handler.
run_one_for	Run the io_context object's event processing loop for a specified duration to execute at most one handler.
run_one_until	Run the io_context object's event processing loop until a specified time to execute at most one handler.
run_until	Run the io_context object's event processing loop until a specified time.
stop	Stop the io_context object's event processing loop.
stopped	Determine whether the io_context object has been stopped.
wrap	(Deprecated: Use asio::bind_executor().) Create a new handler that automatically dispatches the wrapped handler on the io_context.
~io_context	Destructor.

Protected Member Functions

Name	Description
destroy	Destroys all services in the context.

Name	Description
shutdown	Shuts down all services in the context.

Friends

Name	Description
add_service	(Deprecated: Use make_service().) Add a service object to the execution_context.
has_service	Determine if an execution_context contains a specified service type.
make_service	Creates a service object and adds it to the execution_context.
use_service	Obtain the service object corresponding to the given type.

The `io_context` 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_context` 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 `restart()` and `notify_fork()` functions. Calling `restart()` while there are unfinished `run()`, `run_one()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` calls results in undefined behaviour. The `notify_fork()` function should not be called while any `io_context` function, or any function on an I/O object that is associated with the `io_context`, is being called in another thread.

Synchronous and asynchronous operations

Synchronous operations on I/O objects implicitly run the `io_context` object for an individual operation. The `io_context` functions `run()`, `run_one()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` must be called for the `io_context` 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()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` for the `io_context`.

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()`, `run_for()`, `run_until()`, `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()`, `run_for()`, `run_until()`, `poll()` or `poll_one()` call may be restarted *without* the need for an intervening call to `restart()`. This allows the thread to rejoin the `io_context` object's thread pool without impacting any other threads in the pool.

For example:

```
asio::io_context io_context;
...
for (;;)
{
    try
    {
        io_context.run();
        break; // run() exited normally
    }
    catch (my_exception& e)
    {
        // Deal with exception as appropriate.
    }
}
```

Submitting arbitrary tasks to the `io_context`

To submit functions to the `io_context`, use the `dispatch`, `post` or `defer` free functions.

For example:

```
void my_task()
{
    ...
}

...

asio::io_context io_context;

// Submit a function to the io_context.
asio::post(io_context, my_task);

// Submit a lambda object to the io_context.
asio::post(io_context,
    []()
    {
        ...
    });
}

// Run the io_context until it runs out of work.
io_context.run();
```

Stopping the `io_context` from running out of work

Some applications may need to prevent an `io_context` object's `run()` call from returning when there is no more work to do. For example, the `io_context` 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 `asio::executor_work_guard<io_context::executor_type>`:

```
asio::io_context io_context;
asio::executor_work_guard<asio::io_context::executor_type>
    = asio::make_work_guard(io_context);
...
```

To effect a shutdown, the application will then need to call the `io_context` object's `stop()` member function. This will cause the `io_context 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 reset.

```
asio::io_context io_context;
asio::executor_work_guard<asio::io_context::executor_type>
    = asio::make_work_guard(io_context);
...
work.reset(); // Allow run() to exit.
```

Requirements

Header: `asio/io_service.hpp`

Convenience header: `asio.hpp`

5.150 ip::address

Implements version-independent IP addresses.

```
class address
```

Member Functions

Name	Description
<code>address</code>	Default constructor. Construct an address from an IPv4 address. Construct an address from an IPv6 address. Copy constructor.
<code>from_string</code>	(Deprecated: Use <code>make_address()</code>) Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.
<code>is_loopback</code>	Determine whether the address is a loopback address.
<code>is_multicast</code>	Determine whether the address is a multicast address.
<code>is_unspecified</code>	Determine whether the address is unspecified.
<code>is_v4</code>	Get whether the address is an IP version 4 address.
<code>is_v6</code>	Get whether the address is an IP version 6 address.
<code>operator=</code>	Assign from another address. Assign from an IPv4 address. Assign from an IPv6 address.
<code>to_string</code>	Get the address as a string. (Deprecated: Use other overload.) Get the address as a string.

Name	Description
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.
operator>=	Compare addresses for ordering.

Related Functions

Name	Description
make_address	Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.
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`

5.150.1 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);
```

5.150.1.1 ip::address::address (1 of 4 overloads)

Default constructor.

```
address();
```

5.150.1.2 ip::address::address (2 of 4 overloads)

Construct an address from an IPv4 address.

```
address(  
    const asio::ip::address_v4 & ipv4_address);
```

5.150.1.3 ip::address::address (3 of 4 overloads)

Construct an address from an IPv6 address.

```
address(  
    const asio::ip::address_v6 & ipv6_address);
```

5.150.1.4 ip::address::address (4 of 4 overloads)

Copy constructor.

```
address(  
    const address & other);
```

5.150.2 ip::address::from_string

(Deprecated: Use `make_address()`.) 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);
```

5.150.2.1 ip::address::from_string (1 of 4 overloads)

(Deprecated: Use `make_address()`) 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);
```

5.150.2.2 ip::address::from_string (2 of 4 overloads)

(Deprecated: Use `make_address()`) 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);
```

5.150.2.3 ip::address::from_string (3 of 4 overloads)

(Deprecated: Use `make_address()`) 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);
```

5.150.2.4 ip::address::from_string (4 of 4 overloads)

(Deprecated: Use `make_address()`) 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);
```

5.150.3 ip::address::is_loopback

Determine whether the address is a loopback address.

```
bool is_loopback() const;
```

5.150.4 ip::address::is_multicast

Determine whether the address is a multicast address.

```
bool is_multicast() const;
```

5.150.5 ip::address::is_unspecified

Determine whether the address is unspecified.

```
bool is_unspecified() const;
```

5.150.6 ip::address::is_v4

Get whether the address is an IP version 4 address.

```
bool is_v4() const;
```

5.150.7 ip::address::is_v6

Get whether the address is an IP version 6 address.

```
bool is_v6() const;
```

5.150.8 ip::address::make_address

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
address make_address(
    const char * str);

address make_address(
    const char * str,
    asio::error_code & ec);

address make_address(
    const std::string & str);

address make_address(
    const std::string & str,
    asio::error_code & ec);

address make_address(
    string_view str);

address make_address(
    string_view str,
    asio::error_code & ec);
```

5.150.8.1 ip::address::make_address (1 of 6 overloads)

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
address make_address(
    const char * str);
```

5.150.8.2 ip::address::make_address (2 of 6 overloads)

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
address make_address(
    const char * str,
   asio::error_code & ec);
```

5.150.8.3 ip::address::make_address (3 of 6 overloads)

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
address make_address(
    const std::string & str);
```

5.150.8.4 ip::address::make_address (4 of 6 overloads)

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
address make_address(
    const std::string & str,
    asio::error_code & ec);
```

5.150.8.5 ip::address::make_address (5 of 6 overloads)

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
address make_address(
    string_view str);
```

5.150.8.6 ip::address::make_address (6 of 6 overloads)

Create an address from an IPv4 address string in dotted decimal form, or from an IPv6 address in hexadecimal notation.

```
address make_address(
    string_view str,
    asio::error_code & ec);
```

5.150.9 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

5.150.10 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

5.150.11 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.

5.150.12 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

5.150.13 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);
```

5.150.13.1 ip::address::operator= (1 of 3 overloads)

Assign from another address.

```
address & operator=(  
    const address & other);
```

5.150.13.2 ip::address::operator= (2 of 3 overloads)

Assign from an IPv4 address.

```
address & operator=(  
    const asio::ip::address_v4 & ipv4_address);
```

5.150.13.3 ip::address::operator= (3 of 3 overloads)

Assign from an IPv6 address.

```
address & operator=(  
    const asio::ip::address_v6 & ipv6_address);
```

5.150.14 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

5.150.15 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

5.150.16 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

5.150.17 ip::address::to_string

Get the address as a string.

```
std::string to_string() const;
```

(Deprecated: Use other overload.) Get the address as a string.

```
std::string to_string(
    asio::error_code & ec) const;
```

5.150.17.1 ip::address::to_string (1 of 2 overloads)

Get the address as a string.

```
std::string to_string() const;
```

5.150.17.2 ip::address::to_string (2 of 2 overloads)

(Deprecated: Use other overload.) Get the address as a string.

```
std::string to_string(
    asio::error_code & ec) const;
```

5.150.18 ip::address::to_v4

Get the address as an IP version 4 address.

```
asio::ip::address_v4 to_v4() const;
```

5.150.19 ip::address::to_v6

Get the address as an IP version 6 address.

```
asio::ip::address_v6 to_v6() const;
```

5.151 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.
uint_type	The type used to represent an address as an unsigned integer.

Member Functions

Name	Description
address_v4	Default constructor. Construct an address from raw bytes. Construct an address from an unsigned integer 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. (Deprecated: Use network_v4 class.) Obtain an address object that represents the broadcast address that corresponds to the specified address and netmask.
from_string	(Deprecated: Use make_address_v4().) Create an address from an IP address string in dotted decimal form.
is_class_a	(Deprecated: Use network_v4 class.) Determine whether the address is a class A address.

Name	Description
is_class_b	(Deprecated: Use network_v4 class.) Determine whether the address is a class B address.
is_class_c	(Deprecated: Use network_v4 class.) Determine whether the address is a class C address.
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.
loopback	Obtain an address object that represents the loopback address.
netmask	(Deprecated: Use network_v4 class.) 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. (Deprecated: Use other overload.) Get the address as a string in dotted decimal format.
to_uint	Get the address as an unsigned integer in host byte order.
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
make_address_v4	Create an IPv4 address from raw bytes in network order. Create an IPv4 address from an unsigned integer in host byte order. Create an IPv4 address from an IP address string in dotted decimal form. Create an IPv4 address from a IPv4-mapped IPv6 address.
make_network_v4	Create an IPv4 network from an address and prefix length. Create an IPv4 network from an address and netmask.
operator<<	Output an address as a string. Output a network 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`

5.151.1 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 an unsigned integer in host byte order.

```
explicit address_v4(
    uint_type addr);
```

Copy constructor.

```
address_v4(
    const address_v4 & other);
```

5.151.1.1 ip::address_v4::address_v4 (1 of 4 overloads)

Default constructor.

```
address_v4();
```

5.151.1.2 ip::address_v4::address_v4 (2 of 4 overloads)

Construct an address from raw bytes.

```
address_v4(  
    const bytes_type & bytes);
```

5.151.1.3 ip::address_v4::address_v4 (3 of 4 overloads)

Construct an address from an unsigned integer in host byte order.

```
address_v4(  
    uint_type addr);
```

5.151.1.4 ip::address_v4::address_v4 (4 of 4 overloads)

Copy constructor.

```
address_v4(  
    const address_v4 & other);
```

5.151.2 ip::address_v4::any

Obtain an address object that represents any address.

```
static address_v4 any();
```

5.151.3 ip::address_v4::broadcast

Obtain an address object that represents the broadcast address.

```
static address_v4 broadcast();
```

(Deprecated: Use [ip::network_v4](#) class.) 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);
```

5.151.3.1 ip::address_v4::broadcast (1 of 2 overloads)

Obtain an address object that represents the broadcast address.

```
static address_v4 broadcast();
```

5.151.3.2 ip::address_v4::broadcast (2 of 2 overloads)

(Deprecated: Use `ip::network_v4` class.) 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);
```

5.151.4 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`

5.151.5 ip::address_v4::from_string

(Deprecated: Use `make_address_v4()`.) 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);
```

5.151.5.1 ip::address_v4::from_string (1 of 4 overloads)

(Deprecated: Use `make_address_v4()`.) Create an address from an IP address string in dotted decimal form.

```
static address_v4 from_string(
    const char * str);
```

5.151.5.2 ip::address_v4::from_string (2 of 4 overloads)

(Deprecated: Use `make_address_v4()`.) Create an address from an IP address string in dotted decimal form.

```
static address_v4 from_string(
    const char * str,
    asio::error_code & ec);
```

5.151.5.3 ip::address_v4::from_string (3 of 4 overloads)

(Deprecated: Use `make_address_v4()`.) Create an address from an IP address string in dotted decimal form.

```
static address_v4 from_string(
    const std::string & str);
```

5.151.5.4 ip::address_v4::from_string (4 of 4 overloads)

(Deprecated: Use `make_address_v4()`.) 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);
```

5.151.6 ip::address_v4::is_class_a

(Deprecated: Use [ip::network_v4](#) class.) Determine whether the address is a class A address.

```
bool is_class_a() const;
```

5.151.7 ip::address_v4::is_class_b

(Deprecated: Use [ip::network_v4](#) class.) Determine whether the address is a class B address.

```
bool is_class_b() const;
```

5.151.8 ip::address_v4::is_class_c

(Deprecated: Use [ip::network_v4](#) class.) Determine whether the address is a class C address.

```
bool is_class_c() const;
```

5.151.9 ip::address_v4::is_loopback

Determine whether the address is a loopback address.

```
bool is_loopback() const;
```

5.151.10 ip::address_v4::is_multicast

Determine whether the address is a multicast address.

```
bool is_multicast() const;
```

5.151.11 ip::address_v4::is_unspecified

Determine whether the address is unspecified.

```
bool is_unspecified() const;
```

5.151.12 ip::address_v4::loopback

Obtain an address object that represents the loopback address.

```
static address_v4 loopback();
```

5.151.13 ip::address_v4::make_address_v4

Create an IPv4 address from raw bytes in network order.

```
address_v4 make_address_v4(
    const address_v4::bytes_type & bytes);
```

Create an IPv4 address from an unsigned integer in host byte order.

```
address_v4 make_address_v4(
    address_v4::uint_type addr);
```

Create an IPv4 address from an IP address string in dotted decimal form.

```
address_v4 make_address_v4(
    const char * str);
```

```
address_v4 make_address_v4(
    const char * str,
    asio::error_code & ec);
```

```
address_v4 make_address_v4(
    const std::string & str);
```

```
address_v4 make_address_v4(
    const std::string & str,
    asio::error_code & ec);
```

```
address_v4 make_address_v4(
    string_view str);
```

```
address_v4 make_address_v4(
    string_view str,
    asio::error_code & ec);
```

Create an IPv4 address from a IPv4-mapped IPv6 address.

```
address_v4 make_address_v4(
    v4_mapped_t ,
    const address_v6 & v6_addr);
```

5.151.13.1 ip::address_v4::make_address_v4 (1 of 9 overloads)

Create an IPv4 address from raw bytes in network order.

```
address_v4 make_address_v4(  
    const address_v4::bytes_type & bytes);
```

5.151.13.2 ip::address_v4::make_address_v4 (2 of 9 overloads)

Create an IPv4 address from an unsigned integer in host byte order.

```
address_v4 make_address_v4(  
    address_v4::uint_type addr);
```

5.151.13.3 ip::address_v4::make_address_v4 (3 of 9 overloads)

Create an IPv4 address from an IP address string in dotted decimal form.

```
address_v4 make_address_v4(  
    const char * str);
```

5.151.13.4 ip::address_v4::make_address_v4 (4 of 9 overloads)

Create an IPv4 address from an IP address string in dotted decimal form.

```
address_v4 make_address_v4(  
    const char * str,  
    asio::error_code & ec);
```

5.151.13.5 ip::address_v4::make_address_v4 (5 of 9 overloads)

Create an IPv4 address from an IP address string in dotted decimal form.

```
address_v4 make_address_v4(  
    const std::string & str);
```

5.151.13.6 ip::address_v4::make_address_v4 (6 of 9 overloads)

Create an IPv4 address from an IP address string in dotted decimal form.

```
address_v4 make_address_v4(  
    const std::string & str,  
    asio::error_code & ec);
```

5.151.13.7 ip::address_v4::make_address_v4 (7 of 9 overloads)

Create an IPv4 address from an IP address string in dotted decimal form.

```
address_v4 make_address_v4(  
    string_view str);
```

5.151.13.8 ip::address_v4::make_address_v4 (8 of 9 overloads)

Create an IPv4 address from an IP address string in dotted decimal form.

```
address_v4 make_address_v4(  
    string_view str,  
    asio::error_code & ec);
```

5.151.13.9 ip::address_v4::make_address_v4 (9 of 9 overloads)

Create an IPv4 address from a IPv4-mapped IPv6 address.

```
address_v4 make_address_v4 (
    v4_mapped_t ,
    const address_v6 & v6_addr);
```

5.151.14 ip::address_v4::make_network_v4

Create an IPv4 network from an address and prefix length.

```
network_v4 make_network_v4 (
    const address_v4 & addr,
    unsigned short prefix_len);
```

Create an IPv4 network from an address and netmask.

```
network_v4 make_network_v4 (
    const address_v4 & addr,
    const address_v4 & mask);
```

5.151.14.1 ip::address_v4::make_network_v4 (1 of 2 overloads)

Create an IPv4 network from an address and prefix length.

```
network_v4 make_network_v4 (
    const address_v4 & addr,
    unsigned short prefix_len);
```

5.151.14.2 ip::address_v4::make_network_v4 (2 of 2 overloads)

Create an IPv4 network from an address and netmask.

```
network_v4 make_network_v4 (
    const address_v4 & addr,
    const address_v4 & mask);
```

5.151.15 ip::address_v4::netmask

(Deprecated: Use [ip::network_v4](#) class.) Obtain the netmask that corresponds to the address, based on its address class.

```
static address_v4 netmask (
    const address_v4 & addr);
```

5.151.16 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

5.151.17 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

5.151.18 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);
```

Output a network as a string.

```
template<
    typename Elem,
    typename Traits>
std::basic_ostream< Elem, Traits > & operator<<(
    std::basic_ostream< Elem, Traits > & os,
    const network_v4 & net);
```

5.151.18.1 ip::address_v4::operator<< (1 of 2 overloads)

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.

5.151.18.2 ip::address_v4::operator<< (2 of 2 overloads)

Output a network as a string.

```
template<
    typename Elem,
    typename Traits>
std::basic_ostream<Elem, Traits> & operator<<(
    std::basic_ostream<Elem, Traits> & os,
    const network_v4 & net);
```

Used to output a human-readable string for a specified network.

Parameters

os The output stream to which the string will be written.

net The network to be written.

Return Value

The output stream.

5.151.19 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

5.151.20 ip::address_v4::operator=

Assign from another address.

```
address_v4 & operator=(
    const address_v4 & other);
```

5.151.21 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

5.151.22 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

5.151.23 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

5.151.24 ip::address_v4::to_bytes

Get the address in bytes, in network byte order.

```
bytes_type to_bytes() const;
```

5.151.25 ip::address_v4::to_string

Get the address as a string in dotted decimal format.

```
std::string to_string() const;
```

(Deprecated: Use other overload.) Get the address as a string in dotted decimal format.

```
std::string to_string(  
    asio::error_code & ec) const;
```

5.151.25.1 ip::address_v4::to_string (1 of 2 overloads)

Get the address as a string in dotted decimal format.

```
std::string to_string() const;
```

5.151.25.2 ip::address_v4::to_string (2 of 2 overloads)

(Deprecated: Use other overload.) Get the address as a string in dotted decimal format.

```
std::string to_string(  
    asio::error_code & ec) const;
```

5.151.26 ip::address_v4::to_uint

Get the address as an unsigned integer in host byte order.

```
uint_type to_uint() const;
```

5.151.27 ip::address_v4::to_ulong

Get the address as an unsigned long in host byte order.

```
unsigned long to_ulong() const;
```

5.151.28 ip::address_v4::uint_type

The type used to represent an address as an unsigned integer.

```
typedef uint_least32_t uint_type;
```

Requirements

Header: asio/ip/address_v4.hpp

Convenience header: asio.hpp

5.152 ip::address_v4_iterator

An input iterator that can be used for traversing IPv4 addresses.

```
typedef basic_address_iterator< address_v4 > address_v4_iterator;
```

Types

Name	Description
<code>difference_type</code>	Distance between two iterators.
<code>iterator_category</code>	Denotes that the iterator satisfies the input iterator requirements.
<code>pointer</code>	The type of a pointer to an element pointed to by the iterator.
<code>reference</code>	The type of a reference to an element pointed to by the iterator.
<code>value_type</code>	The type of the elements pointed to by the iterator.

Member Functions

Name	Description
<code>basic_address_iterator</code>	Construct an iterator that points to the specified address. Copy constructor.
<code>operator *</code>	Dereference the iterator.
<code>operator++</code>	Pre-increment operator. Post-increment operator.
<code>operator--</code>	Pre-decrement operator. Post-decrement operator.
<code>operator-></code>	Dereference the iterator.
<code>operator=</code>	Assignment operator.

Friends

Name	Description
<code>operator!=</code>	Compare two addresses for inequality.
<code>operator==</code>	Compare two addresses for equality.

In addition to satisfying the input iterator requirements, this iterator also supports decrement.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v4_iterator.hpp

Convenience header: asio.hpp

5.153 ip::address_v4_range

Represents a range of IPv4 addresses.

```
typedef basic_address_range< address_v4 > address_v4_range;
```

Types

Name	Description
iterator	The type of an iterator that points into the range.

Member Functions

Name	Description
basic_address_range	Construct an empty range. Construct a range that represents the given range of addresses. Copy constructor.
begin	Obtain an iterator that points to the start of the range.
empty	Determine whether the range is empty.
end	Obtain an iterator that points to the end of the range.
find	Find an address in the range.
operator=	Assignment operator.
size	Return the size of the range.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v4_range.hpp

Convenience header: asio.hpp

5.154 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.
any	Obtain an address object that represents any address.
from_string	(Deprecated: Use make_address_v6().) Create an IPv6 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	(Deprecated: No replacement.) Determine whether the address is an IPv4-compatible address.
is_v4_mapped	Determine whether the address is a mapped IPv4 address.

Name	Description
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. (Deprecated: Use other overload.) Get the address as a string.
to_v4	(Deprecated: Use make_address_v4().) Converts an IPv4-mapped or IPv4-compatible address to an IPv4 address.
v4_compatible	(Deprecated: No replacement.) Create an IPv4-compatible IPv6 address.
v4_mapped	(Deprecated: Use make_address_v6().) 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
make_address_v6	Create an IPv6 address from raw bytes and scope ID. Create an IPv6 address from an IP address string. Create an IPv6 address from an IP address string. Create an IPv4-mapped IPv6 address from an IPv4 address.
make_network_v6	Create an IPv6 network from an address and prefix length.
operator<<	Output an address as a string. Output a network 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`

5.154.1 `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);
```

5.154.1.1 `ip::address_v6::address_v6 (1 of 3 overloads)`

Default constructor.

```
address_v6();
```

5.154.1.2 `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);
```

5.154.1.3 `ip::address_v6::address_v6 (3 of 3 overloads)`

Copy constructor.

```
address_v6(
    const address_v6 & other);
```

5.154.2 ip::address_v6::any

Obtain an address object that represents any address.

```
static address_v6 any();
```

5.154.3 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`

5.154.4 ip::address_v6::from_string

(Deprecated: Use `make_address_v6()`.) Create an IPv6 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);
```

5.154.4.1 ip::address_v6::from_string (1 of 4 overloads)

(Deprecated: Use `make_address_v6()`.) Create an IPv6 address from an IP address string.

```
static address_v6 from_string(
    const char * str);
```

5.154.4.2 ip::address_v6::from_string (2 of 4 overloads)

(Deprecated: Use `make_address_v6()`.) Create an IPv6 address from an IP address string.

```
static address_v6 from_string(
    const char * str,
    asio::error_code & ec);
```

5.154.4.3 ip::address_v6::from_string (3 of 4 overloads)

(Deprecated: Use `make_address_v6()`.) Create an IPv6 address from an IP address string.

```
static address_v6 from_string(
    const std::string & str);
```

5.154.4.4 ip::address_v6::from_string (4 of 4 overloads)

(Deprecated: Use `make_address_v6()`.) Create an IPv6 address from an IP address string.

```
static address_v6 from_string(
    const std::string & str,
    asio::error_code & ec);
```

5.154.5 ip::address_v6::is_link_local

Determine whether the address is link local.

```
bool is_link_local() const;
```

5.154.6 ip::address_v6::is_loopback

Determine whether the address is a loopback address.

```
bool is_loopback() const;
```

5.154.7 ip::address_v6::is_multicast

Determine whether the address is a multicast address.

```
bool is_multicast() const;
```

5.154.8 ip::address_v6::is_multicast_global

Determine whether the address is a global multicast address.

```
bool is_multicast_global() const;
```

5.154.9 ip::address_v6::is_multicast_link_local

Determine whether the address is a link-local multicast address.

```
bool is_multicast_link_local() const;
```

5.154.10 ip::address_v6::is_multicast_node_local

Determine whether the address is a node-local multicast address.

```
bool is_multicast_node_local() const;
```

5.154.11 ip::address_v6::is_multicast_org_local

Determine whether the address is a org-local multicast address.

```
bool is_multicast_org_local() const;
```

5.154.12 ip::address_v6::is_multicast_site_local

Determine whether the address is a site-local multicast address.

```
bool is_multicast_site_local() const;
```

5.154.13 ip::address_v6::is_site_local

Determine whether the address is site local.

```
bool is_site_local() const;
```

5.154.14 ip::address_v6::is_unspecified

Determine whether the address is unspecified.

```
bool is_unspecified() const;
```

5.154.15 ip::address_v6::is_v4_compatible

(Deprecated: No replacement.) Determine whether the address is an IPv4-compatible address.

```
bool is_v4_compatible() const;
```

5.154.16 ip::address_v6::is_v4_mapped

Determine whether the address is a mapped IPv4 address.

```
bool is_v4_mapped() const;
```

5.154.17 ip::address_v6::loopback

Obtain an address object that represents the loopback address.

```
static address_v6 loopback();
```

5.154.18 ip::address_v6::make_address_v6

Create an IPv6 address from raw bytes and scope ID.

```
address_v6 make_address_v6(
    const address_v6::bytes_type & bytes,
    unsigned long scope_id = 0);
```

Create an IPv6 address from an IP address string.

```
address_v6 make_address_v6(
    const char * str);
```

```
address_v6 make_address_v6(
    const char * str,
    asio::error_code & ec);
```

Create an IPv6 address from an IP address string.

```
address_v6 make_address_v6(
    const std::string & str);
```

```
address_v6 make_address_v6(
    const std::string & str,
    asio::error_code & ec);
```

```
address_v6 make_address_v6(
    string_view str);
```

```
address_v6 make_address_v6(
    string_view str,
    asio::error_code & ec);
```

Create an IPv4-mapped IPv6 address from an IPv4 address.

```
address_v6 make_address_v6(
    v4_mapped_t ,
    const address_v4 & v4_addr);
```

5.154.18.1 ip::address_v6::make_address_v6 (1 of 8 overloads)

Create an IPv6 address from raw bytes and scope ID.

```
address_v6 make_address_v6(
    const address_v6::bytes_type & bytes,
    unsigned long scope_id = 0);
```

5.154.18.2 ip::address_v6::make_address_v6 (2 of 8 overloads)

Create an IPv6 address from an IP address string.

```
address_v6 make_address_v6(
    const char * str);
```

5.154.18.3 ip::address_v6::make_address_v6 (3 of 8 overloads)

Create an IPv6 address from an IP address string.

```
address_v6 make_address_v6(
    const char * str,
   asio::error_code & ec);
```

5.154.18.4 ip::address_v6::make_address_v6 (4 of 8 overloads)

Create an IPv6 address from an IP address string.

```
address_v6 make_address_v6(
    const std::string & str);
```

5.154.18.5 ip::address_v6::make_address_v6 (5 of 8 overloads)

Create an IPv6 address from an IP address string.

```
address_v6 make_address_v6(
    const std::string & str,
    asio::error_code & ec);
```

5.154.18.6 ip::address_v6::make_address_v6 (6 of 8 overloads)

Create an IPv6 address from an IP address string.

```
address_v6 make_address_v6(
    string_view str);
```

5.154.18.7 ip::address_v6::make_address_v6 (7 of 8 overloads)

Create an IPv6 address from an IP address string.

```
address_v6 make_address_v6(
    string_view str,
    asio::error_code & ec);
```

5.154.18.8 ip::address_v6::make_address_v6 (8 of 8 overloads)

Create an IPv4-mapped IPv6 address from an IPv4 address.

```
address_v6 make_address_v6(
    v4_mapped_t ,
    const address_v4 & v4_addr);
```

5.154.19 ip::address_v6::make_network_v6

Create an IPv6 network from an address and prefix length.

```
network_v6 make_network_v6(
    const address_v6 & addr,
    unsigned short prefix_len);
```

5.154.20 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

5.154.21 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

5.154.22 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);
```

Output a network as a string.

```
template<
    typename Elem,
    typename Traits>
std::basic_ostream<Elem, Traits> & operator<<(
    std::basic_ostream<Elem, Traits> & os,
    const network_v6 & net);
```

5.154.22.1 ip::address_v6::operator<< (1 of 2 overloads)

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.

5.154.22.2 ip::address_v6::operator<< (2 of 2 overloads)

Output a network as a string.

```
template<
    typename Elem,
    typename Traits>
std::basic_ostream< Elem, Traits > & operator<<(
    std::basic_ostream< Elem, Traits > & os,
    const network_v6 & net);
```

Used to output a human-readable string for a specified network.

Parameters

os The output stream to which the string will be written.

net The network to be written.

Return Value

The output stream.

5.154.23 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

5.154.24 ip::address_v6::operator=

Assign from another address.

```
address_v6 & operator=(  
    const address_v6 & other);
```

5.154.25 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

5.154.26 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

5.154.27 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

5.154.28 ip::address_v6::scope_id

The scope ID of the address.

```
unsigned long scope_id() const;  
  
void scope_id(  
    unsigned long id);
```

5.154.28.1 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.

5.154.28.2 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.

5.154.29 ip::address_v6::to_bytes

Get the address in bytes, in network byte order.

```
bytes_type to_bytes() const;
```

5.154.30 ip::address_v6::to_string

Get the address as a string.

```
std::string to_string() const;
```

(Deprecated: Use other overload.) Get the address as a string.

```
std::string to_string(  
    asio::error_code & ec) const;
```

5.154.30.1 ip::address_v6::to_string (1 of 2 overloads)

Get the address as a string.

```
std::string to_string() const;
```

5.154.30.2 ip::address_v6::to_string (2 of 2 overloads)

(Deprecated: Use other overload.) Get the address as a string.

```
std::string to_string(
    asio::error_code & ec) const;
```

5.154.31 ip::address_v6::to_v4

(Deprecated: Use make_address_v4 () .) Converts an IPv4-mapped or IPv4-compatible address to an IPv4 address.

```
address_v4 to_v4() const;
```

5.154.32 ip::address_v6::v4_compatible

(Deprecated: No replacement.) Create an IPv4-compatible IPv6 address.

```
static address_v6 v4_compatible(
    const address_v4 & addr);
```

5.154.33 ip::address_v6::v4_mapped

(Deprecated: Use make_address_v6 () .) Create an IPv4-mapped IPv6 address.

```
static address_v6 v4_mapped(
    const address_v4 & addr);
```

5.155 ip::address_v6_iterator

An input iterator that can be used for traversing IPv6 addresses.

```
typedef basic_address_iterator< address_v6 > address_v6_iterator;
```

Types

Name	Description
difference_type	Distance between two iterators.
iterator_category	Denotes that the iterator satisfies the input iterator requirements.
pointer	The type of a pointer to an element pointed to by the iterator.
reference	The type of a reference to an element pointed to by the iterator.
value_type	The type of the elements pointed to by the iterator.

Member Functions

Name	Description
basic_address_iterator	Construct an iterator that points to the specified address. Copy constructor.
operator *	Dereference the iterator.
operator++	Pre-increment operator. Post-increment operator.
operator--	Pre-decrement operator. Post-decrement operator.
operator->	Dereference the iterator.
operator=	Assignment operator.

Friends

Name	Description
operator!=	Compare two addresses for inequality.
operator==	Compare two addresses for equality.

In addition to satisfying the input iterator requirements, this iterator also supports decrement.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v6_iterator.hpp

Convenience header: asio.hpp

5.156 ip::address_v6_range

Represents a range of IPv6 addresses.

```
typedef basic_address_range< address_v6 > address_v6_range;
```

Types

Name	Description
iterator	The type of an iterator that points into the range.

Member Functions

Name	Description
basic_address_range	Construct an empty range. Construct a range that represents the given range of addresses. Copy constructor.
begin	Obtain an iterator that points to the start of the range.
empty	Determine whether the range is empty.
end	Obtain an iterator that points to the end of the range.
find	Find an address in the range.
operator=	Assignment operator.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v6_range.hpp

Convenience header: asio.hpp

5.157 ip::bad_address_cast

Thrown to indicate a failed address conversion.

```
class bad_address_cast
```

Member Functions

Name	Description
bad_address_cast	Default constructor.
what	Get the message associated with the exception.
~bad_address_cast	Destructor.

Requirements

Header: asio/ip/bad_address_cast.hpp

Convenience header: asio.hpp

5.157.1 ip::bad_address_cast::bad_address_cast

Default constructor.

```
bad_address_cast();
```

5.157.2 ip::bad_address_cast::what

Get the message associated with the exception.

```
virtual const char * what() const;
```

5.157.3 ip::bad_address_cast::~bad_address_cast

Destructor.

```
virtual ~bad_address_cast();
```

5.158 ip::basic_address_iterator< address_v4 >

An input iterator that can be used for traversing IPv4 addresses.

```
template<>
class basic_address_iterator< address_v4 >
```

Types

Name	Description
difference_type	Distance between two iterators.
iterator_category	Denotes that the iterator satisfies the input iterator requirements.
pointer	The type of a pointer to an element pointed to by the iterator.
reference	The type of a reference to an element pointed to by the iterator.
value_type	The type of the elements pointed to by the iterator.

Member Functions

Name	Description
basic_address_iterator	Construct an iterator that points to the specified address. Copy constructor.
operator *	Dereference the iterator.

Name	Description
operator++	Pre-increment operator. Post-increment operator.
operator--	Pre-decrement operator. Post-decrement operator.
operator->	Dereference the iterator.
operator=	Assignment operator.

Friends

Name	Description
operator!=	Compare two addresses for inequality.
operator==	Compare two addresses for equality.

In addition to satisfying the input iterator requirements, this iterator also supports decrement.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v4_iterator.hpp

Convenience header: asio.hpp

5.158.1 ip::basic_address_iterator< address_v4 >::basic_address_iterator

Construct an iterator that points to the specified address.

```
basic_address_iterator(
    const address_v4 & addr);
```

Copy constructor.

```
basic_address_iterator(
    const basic_address_iterator & other);
```

5.158.1.1 ip::basic_address_iterator< address_v4 >::basic_address_iterator (1 of 2 overloads)

Construct an iterator that points to the specified address.

```
basic_address_iterator(
    const address_v4 & addr);
```

5.158.1.2 ip::basic_address_iterator< address_v4 >::basic_address_iterator (2 of 2 overloads)

Copy constructor.

```
basic_address_iterator(
    const basic_address_iterator & other);
```

5.158.2 ip::basic_address_iterator< address_v4 >::difference_type

Distance between two iterators.

```
typedef std::ptrdiff_t difference_type;
```

Requirements

Header: asio/ip/address_v4_iterator.hpp

Convenience header: asio.hpp

5.158.3 ip::basic_address_iterator< address_v4 >::iterator_category

Denotes that the iterator satisfies the input iterator requirements.

```
typedef std::input_iterator_tag iterator_category;
```

Requirements

Header: asio/ip/address_v4_iterator.hpp

Convenience header: asio.hpp

5.158.4 ip::basic_address_iterator< address_v4 >::operator *

Dereference the iterator.

```
const address_v4 & operator *() const;
```

5.158.5 ip::basic_address_iterator< address_v4 >::operator!=

Compare two addresses for inequality.

```
friend bool operator!=(
    const basic_address_iterator & a,
    const basic_address_iterator & b);
```

Requirements

Header: asio/ip/address_v4_iterator.hpp

Convenience header: asio.hpp

5.158.6 ip::basic_address_iterator< address_v4 >::operator++

Pre-increment operator.

```
basic_address_iterator & operator++();
```

Post-increment operator.

```
basic_address_iterator operator++( int );
```

5.158.6.1 ip::basic_address_iterator< address_v4 >::operator++ (1 of 2 overloads)

Pre-increment operator.

```
basic_address_iterator & operator++();
```

5.158.6.2 ip::basic_address_iterator< address_v4 >::operator++ (2 of 2 overloads)

Post-increment operator.

```
basic_address_iterator operator++( int );
```

5.158.7 ip::basic_address_iterator< address_v4 >::operator--

Pre-decrement operator.

```
basic_address_iterator & operator--();
```

Post-decrement operator.

```
basic_address_iterator operator--( int );
```

5.158.7.1 ip::basic_address_iterator< address_v4 >::operator-- (1 of 2 overloads)

Pre-decrement operator.

```
basic_address_iterator & operator--();
```

5.158.7.2 ip::basic_address_iterator< address_v4 >::operator-- (2 of 2 overloads)

Post-decrement operator.

```
basic_address_iterator operator--( int );
```

5.158.8 ip::basic_address_iterator< address_v4 >::operator->

Dereference the iterator.

```
const address_v4 * operator->() const;
```

5.158.9 ip::basic_address_iterator< address_v4 >::operator=

Assignment operator.

```
basic_address_iterator & operator=(  
    const basic_address_iterator & other);
```

5.158.10 ip::basic_address_iterator< address_v4 >::operator==

Compare two addresses for equality.

```
friend bool operator==(  
    const basic_address_iterator & a,  
    const basic_address_iterator & b);
```

Requirements

Header: asio/ip/address_v4_iterator.hpp

Convenience header: asio.hpp

5.158.11 ip::basic_address_iterator< address_v4 >::pointer

The type of a pointer to an element pointed to by the iterator.

```
typedef const address_v4 * pointer;
```

Requirements

Header: asio/ip/address_v4_iterator.hpp

Convenience header: asio.hpp

5.158.12 ip::basic_address_iterator< address_v4 >::reference

The type of a reference to an element pointed to by the iterator.

```
typedef const address_v4 & reference;
```

Types

Name	Description
bytes_type	The type used to represent an address as an array of bytes.
uint_type	The type used to represent an address as an unsigned integer.

Member Functions

Name	Description
address_v4	Default constructor. Construct an address from raw bytes. Construct an address from an unsigned integer 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. (Deprecated: Use network_v4 class.) Obtain an address object that represents the broadcast address that corresponds to the specified address and netmask.
from_string	(Deprecated: Use make_address_v4().) Create an address from an IP address string in dotted decimal form.
is_class_a	(Deprecated: Use network_v4 class.) Determine whether the address is a class A address.
is_class_b	(Deprecated: Use network_v4 class.) Determine whether the address is a class B address.
is_class_c	(Deprecated: Use network_v4 class.) Determine whether the address is a class C address.
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.
loopback	Obtain an address object that represents the loopback address.
netmask	(Deprecated: Use network_v4 class.) 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. (Deprecated: Use other overload.) Get the address as a string in dotted decimal format.
to_uint	Get the address as an unsigned integer in host byte order.
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
make_address_v4	Create an IPv4 address from raw bytes in network order. Create an IPv4 address from an unsigned integer in host byte order. Create an IPv4 address from an IP address string in dotted decimal form. Create an IPv4 address from a IPv4-mapped IPv6 address.
make_network_v4	Create an IPv4 network from an address and prefix length. Create an IPv4 network from an address and netmask.
operator<<	Output an address as a string. Output a network 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_iterator.hpp`

Convenience header: `asio.hpp`

5.158.13 ip::basic_address_iterator< address_v4 >::value_type

The type of the elements pointed to by the iterator.

```
typedef address_v4 value_type;
```

Types

Name	Description
bytes_type	The type used to represent an address as an array of bytes.
uint_type	The type used to represent an address as an unsigned integer.

Member Functions

Name	Description
address_v4	Default constructor. Construct an address from raw bytes. Construct an address from an unsigned integer 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. (Deprecated: Use network_v4 class.) Obtain an address object that represents the broadcast address that corresponds to the specified address and netmask.
from_string	(Deprecated: Use make_address_v4().) Create an address from an IP address string in dotted decimal form.
is_class_a	(Deprecated: Use network_v4 class.) Determine whether the address is a class A address.
is_class_b	(Deprecated: Use network_v4 class.) Determine whether the address is a class B address.
is_class_c	(Deprecated: Use network_v4 class.) Determine whether the address is a class C address.
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.
loopback	Obtain an address object that represents the loopback address.
netmask	(Deprecated: Use network_v4 class.) 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. (Deprecated: Use other overload.) Get the address as a string in dotted decimal format.

Name	Description
to_uint	Get the address as an unsigned integer in host byte order.
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
make_address_v4	Create an IPv4 address from raw bytes in network order. Create an IPv4 address from an unsigned integer in host byte order. Create an IPv4 address from an IP address string in dotted decimal form. Create an IPv4 address from a IPv4-mapped IPv6 address.
make_network_v4	Create an IPv4 network from an address and prefix length. Create an IPv4 network from an address and netmask.
operator<<	Output an address as a string. Output a network 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_iterator.hpp`

Convenience header: `asio.hpp`

5.159 ip::basic_address_iterator< address_v6 >

An input iterator that can be used for traversing IPv6 addresses.

```
template<>
class basic_address_iterator< address_v6 >
```

Types

Name	Description
difference_type	Distance between two iterators.
iterator_category	Denotes that the iterator satisfies the input iterator requirements.
pointer	The type of a pointer to an element pointed to by the iterator.
reference	The type of a reference to an element pointed to by the iterator.
value_type	The type of the elements pointed to by the iterator.

Member Functions

Name	Description
basic_address_iterator	Construct an iterator that points to the specified address. Copy constructor.
operator *	Dereference the iterator.
operator++	Pre-increment operator. Post-increment operator.
operator--	Pre-decrement operator. Post-decrement operator.
operator->	Dereference the iterator.
operator=	Assignment operator.

Friends

Name	Description
operator!=	Compare two addresses for inequality.
operator==	Compare two addresses for equality.

In addition to satisfying the input iterator requirements, this iterator also supports decrement.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v6_iterator.hpp

Convenience header: asio.hpp

5.159.1 ip::basic_address_iterator< address_v6 >::basic_address_iterator

Construct an iterator that points to the specified address.

```
basic_address_iterator(  
    const address_v6 & addr);
```

Copy constructor.

```
basic_address_iterator(  
    const basic_address_iterator & other);
```

5.159.1.1 ip::basic_address_iterator< address_v6 >::basic_address_iterator (1 of 2 overloads)

Construct an iterator that points to the specified address.

```
basic_address_iterator(  
    const address_v6 & addr);
```

5.159.1.2 ip::basic_address_iterator< address_v6 >::basic_address_iterator (2 of 2 overloads)

Copy constructor.

```
basic_address_iterator(  
    const basic_address_iterator & other);
```

5.159.2 ip::basic_address_iterator< address_v6 >::difference_type

Distance between two iterators.

```
typedef std::ptrdiff_t difference_type;
```

Requirements

Header: asio/ip/address_v6_iterator.hpp

Convenience header: asio.hpp

5.159.3 ip::basic_address_iterator< address_v6 >::iterator_category

Denotes that the iterator satisfies the input iterator requirements.

```
typedef std::input_iterator_tag iterator_category;
```

Requirements

Header: asio/ip/address_v6_iterator.hpp

Convenience header: asio.hpp

5.159.4 ip::basic_address_iterator< address_v6 >::operator *

Dereference the iterator.

```
const address_v6 & operator *() const;
```

5.159.5 ip::basic_address_iterator< address_v6 >::operator!=

Compare two addresses for inequality.

```
friend bool operator!=(
    const basic_address_iterator & a,
    const basic_address_iterator & b);
```

Requirements

Header: asio/ip/address_v6_iterator.hpp

Convenience header: asio.hpp

5.159.6 ip::basic_address_iterator< address_v6 >::operator++

Pre-increment operator.

```
basic_address_iterator & operator++();
```

Post-increment operator.

```
basic_address_iterator operator++(
    int );
```

5.159.6.1 ip::basic_address_iterator< address_v6 >::operator++ (1 of 2 overloads)

Pre-increment operator.

```
basic_address_iterator & operator++();
```

5.159.6.2 ip::basic_address_iterator< address_v6 >::operator++ (2 of 2 overloads)

Post-increment operator.

```
basic_address_iterator operator++(  
    int );
```

5.159.7 ip::basic_address_iterator< address_v6 >::operator--

Pre-decrement operator.

```
basic_address_iterator & operator--();
```

Post-decrement operator.

```
basic_address_iterator operator--(  
    int );
```

5.159.7.1 ip::basic_address_iterator< address_v6 >::operator-- (1 of 2 overloads)

Pre-decrement operator.

```
basic_address_iterator & operator--();
```

5.159.7.2 ip::basic_address_iterator< address_v6 >::operator-- (2 of 2 overloads)

Post-decrement operator.

```
basic_address_iterator operator--(  
    int );
```

5.159.8 ip::basic_address_iterator< address_v6 >::operator->

Dereference the iterator.

```
const address_v6 * operator->() const;
```

5.159.9 ip::basic_address_iterator< address_v6 >::operator=

Assignment operator.

```
basic_address_iterator & operator=(  
    const basic_address_iterator & other);
```

5.159.10 ip::basic_address_iterator< address_v6 >::operator==

Compare two addresses for equality.

```
friend bool operator==(  
    const basic_address_iterator & a,  
    const basic_address_iterator & b);
```

Requirements

Header: asio/ip/address_v6_iterator.hpp

Convenience header: asio.hpp

5.159.11 ip::basic_address_iterator< address_v6 >::pointer

The type of a pointer to an element pointed to by the iterator.

```
typedef const address_v6 * pointer;
```

Requirements

Header: asio/ip/address_v6_iterator.hpp

Convenience header: asio.hpp

5.159.12 ip::basic_address_iterator< address_v6 >::reference

The type of a reference to an element pointed to by the iterator.

```
typedef const address_v6 & reference;
```

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.
any	Obtain an address object that represents any address.
from_string	(Deprecated: Use make_address_v6().) Create an IPv6 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.

Name	Description
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	(Deprecated: No replacement.) 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. (Deprecated: Use other overload.) Get the address as a string.
to_v4	(Deprecated: Use make_address_v4().) Converts an IPv4-mapped or IPv4-compatible address to an IPv4 address.
v4_compatible	(Deprecated: No replacement.) Create an IPv4-compatible IPv6 address.
v4_mapped	(Deprecated: Use make_address_v6().) 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.

Name	Description
operator>	Compare addresses for ordering.
operator>=	Compare addresses for ordering.

Related Functions

Name	Description
make_address_v6	Create an IPv6 address from raw bytes and scope ID. Create an IPv6 address from an IP address string. Create an IPv6 address from an IP address string. Create an IPv4-mapped IPv6 address from an IPv4 address.
make_network_v6	Create an IPv6 network from an address and prefix length.
operator<<	Output an address as a string. Output a network 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_iterator.hpp`

Convenience header: `asio.hpp`

5.159.13 `ip::basic_address_iterator< address_v6 >::value_type`

The type of the elements pointed to by the iterator.

```
typedef address_v6 value_type;
```

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.
any	Obtain an address object that represents any address.
from_string	(Deprecated: Use make_address_v6().) Create an IPv6 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	(Deprecated: No replacement.) 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. (Deprecated: Use other overload.) Get the address as a string.
to_v4	(Deprecated: Use make_address_v4().) Converts an IPv4-mapped or IPv4-compatible address to an IPv4 address.
v4_compatible	(Deprecated: No replacement.) Create an IPv4-compatible IPv6 address.

Name	Description
v4_mapped	(Deprecated: Use make_address_v6().) 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
make_address_v6	Create an IPv6 address from raw bytes and scope ID. Create an IPv6 address from an IP address string. Create an IPv6 address from an IP address string. Create an IPv4-mapped IPv6 address from an IPv4 address.
make_network_v6	Create an IPv6 network from an address and prefix length.
operator<<	Output an address as a string. Output a network 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_iterator.hpp`

Convenience header: `asio.hpp`

5.160 ip::basic_address_range< address_v4 >

Represents a range of IPv4 addresses.

```
template<>
class basic_address_range< address_v4 >
```

Types

Name	Description
iterator	The type of an iterator that points into the range.

Member Functions

Name	Description
basic_address_range	Construct an empty range. Construct a range that represents the given range of addresses. Copy constructor.
begin	Obtain an iterator that points to the start of the range.
empty	Determine whether the range is empty.
end	Obtain an iterator that points to the end of the range.
find	Find an address in the range.
operator=	Assignment operator.
size	Return the size of the range.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v4_range.hpp

Convenience header: asio.hpp

5.160.1 ip::basic_address_range< address_v4 >::basic_address_range

Construct an empty range.

```
basic_address_range();
```

Construct an range that represents the given range of addresses.

```
explicit basic_address_range(
    const iterator & first,
    const iterator & last);
```

Copy constructor.

```
basic_address_range(
    const basic_address_range & other);
```

5.160.1.1 ip::basic_address_range< address_v4 >::basic_address_range (1 of 3 overloads)

Construct an empty range.

```
basic_address_range();
```

5.160.1.2 ip::basic_address_range< address_v4 >::basic_address_range (2 of 3 overloads)

Construct an range that represents the given range of addresses.

```
basic_address_range(
    const iterator & first,
    const iterator & last);
```

5.160.1.3 ip::basic_address_range< address_v4 >::basic_address_range (3 of 3 overloads)

Copy constructor.

```
basic_address_range(
    const basic_address_range & other);
```

5.160.2 ip::basic_address_range< address_v4 >::begin

Obtain an iterator that points to the start of the range.

```
iterator begin() const;
```

5.160.3 ip::basic_address_range< address_v4 >::empty

Determine whether the range is empty.

```
bool empty() const;
```

5.160.4 ip::basic_address_range< address_v4 >::end

Obtain an iterator that points to the end of the range.

```
iterator end() const;
```

5.160.5 ip::basic_address_range< address_v4 >::find

Find an address in the range.

```
iterator find(  
    const address_v4 & addr) const;
```

5.160.6 ip::basic_address_range< address_v4 >::iterator

The type of an iterator that points into the range.

```
typedef basic_address_iterator< address_v4 > iterator;
```

Types

Name	Description
difference_type	Distance between two iterators.
iterator_category	Denotes that the iterator satisfies the input iterator requirements.
pointer	The type of a pointer to an element pointed to by the iterator.
reference	The type of a reference to an element pointed to by the iterator.
value_type	The type of the elements pointed to by the iterator.

Member Functions

Name	Description
basic_address_iterator	Construct an iterator that points to the specified address. Copy constructor.
operator *	Dereference the iterator.
operator++	Pre-increment operator. Post-increment operator.
operator--	Pre-decrement operator. Post-decrement operator.
operator->	Dereference the iterator.
operator=	Assignment operator.

Friends

Name	Description
operator!=	Compare two addresses for inequality.
operator==	Compare two addresses for equality.

In addition to satisfying the input iterator requirements, this iterator also supports decrement.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v4_range.hpp

Convenience header: asio.hpp

5.160.7 ip::basic_address_range< address_v4 >::operator=

Assignment operator.

```
basic_address_range & operator=(  
    const basic_address_range & other);
```

5.160.8 ip::basic_address_range< address_v4 >::size

Return the size of the range.

```
std::size_t size() const;
```

5.161 ip::basic_address_range< address_v6 >

Represents a range of IPv6 addresses.

```
template<>  
class basic_address_range< address_v6 >
```

Types

Name	Description
iterator	The type of an iterator that points into the range.

Member Functions

Name	Description
basic_address_range	Construct an empty range. Construct a range that represents the given range of addresses. Copy constructor.
begin	Obtain an iterator that points to the start of the range.
empty	Determine whether the range is empty.
end	Obtain an iterator that points to the end of the range.
find	Find an address in the range.
operator=	Assignment operator.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v6_range.hpp

Convenience header: asio.hpp

5.161.1 ip::basic_address_range< address_v6 >::basic_address_range

Construct an empty range.

```
basic_address_range();
```

Construct a range that represents the given range of addresses.

```
explicit basic_address_range(
    const iterator & first,
    const iterator & last);
```

Copy constructor.

```
basic_address_range(
    const basic_address_range & other);
```

5.161.1.1 ip::basic_address_range< address_v6 >::basic_address_range (1 of 3 overloads)

Construct an empty range.

```
basic_address_range();
```

5.161.1.2 ip::basic_address_range< address_v6 >::basic_address_range (2 of 3 overloads)

Construct an range that represents the given range of addresses.

```
basic_address_range(
    const iterator & first,
    const iterator & last);
```

5.161.1.3 ip::basic_address_range< address_v6 >::basic_address_range (3 of 3 overloads)

Copy constructor.

```
basic_address_range(
    const basic_address_range & other);
```

5.161.2 ip::basic_address_range< address_v6 >::begin

Obtain an iterator that points to the start of the range.

```
iterator begin() const;
```

5.161.3 ip::basic_address_range< address_v6 >::empty

Determine whether the range is empty.

```
bool empty() const;
```

5.161.4 ip::basic_address_range< address_v6 >::end

Obtain an iterator that points to the end of the range.

```
iterator end() const;
```

5.161.5 ip::basic_address_range< address_v6 >::find

Find an address in the range.

```
iterator find(
    const address_v6 & addr) const;
```

5.161.6 ip::basic_address_range< address_v6 >::iterator

The type of an iterator that points into the range.

```
typedef basic_address_iterator< address_v6 > iterator;
```

Types

Name	Description
<code>difference_type</code>	Distance between two iterators.
<code>iterator_category</code>	Denotes that the iterator satisfies the input iterator requirements.
<code>pointer</code>	The type of a pointer to an element pointed to by the iterator.
<code>reference</code>	The type of a reference to an element pointed to by the iterator.
<code>value_type</code>	The type of the elements pointed to by the iterator.

Member Functions

Name	Description
<code>basic_address_iterator</code>	Construct an iterator that points to the specified address. Copy constructor.
<code>operator *</code>	Dereference the iterator.
<code>operator++</code>	Pre-increment operator. Post-increment operator.
<code>operator--</code>	Pre-decrement operator. Post-decrement operator.
<code>operator-></code>	Dereference the iterator.
<code>operator=</code>	Assignment operator.

Friends

Name	Description
<code>operator!=</code>	Compare two addresses for inequality.
<code>operator==</code>	Compare two addresses for equality.

In addition to satisfying the input iterator requirements, this iterator also supports decrement.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/address_v6_range.hpp

Convenience header: asio.hpp

5.161.7 ip::basic_address_range< address_v6 >::operator=

Assignment operator.

```
basic_address_range & operator=(  
    const basic_address_range & other);
```

5.162 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. Move constructor.
capacity	Get the capacity of the endpoint in the native type.

Name	Description
data	Get the underlying endpoint in the native type.
operator=	Assign from another endpoint. Move-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/basic_endpoint.hpp`

Convenience header: asio.hpp

5.162.1 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);
```

5.162.1.1 ip::basic_endpoint::address (1 of 2 overloads)

Get the IP address associated with the endpoint.

```
asio::ip::address address() const;
```

5.162.1.2 ip::basic_endpoint::address (2 of 2 overloads)

Set the IP address associated with the endpoint.

```
void address(
    const asio::ip::address & addr);
```

5.162.2 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. INADDR_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);
```

Move constructor.

```
basic_endpoint(
    basic_endpoint && other);
```

5.162.2.1 ip::basic_endpoint::basic_endpoint (1 of 5 overloads)

Default constructor.

```
basic_endpoint();
```

5.162.2.2 ip::basic_endpoint::basic_endpoint (2 of 5 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);
```

5.162.2.3 ip::basic_endpoint::basic_endpoint (3 of 5 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);
```

5.162.2.4 ip::basic_endpoint::basic_endpoint (4 of 5 overloads)

Copy constructor.

```
basic_endpoint(
    const basic_endpoint & other);
```

5.162.2.5 ip::basic_endpoint::basic_endpoint (5 of 5 overloads)

Move constructor.

```
basic_endpoint(
    basic_endpoint && other);
```

5.162.3 ip::basic_endpoint::capacity

Get the capacity of the endpoint in the native type.

```
std::size_t capacity() const;
```

5.162.4 ip::basic_endpoint::data

Get the underlying endpoint in the native type.

```
data_type * data();  
  
const data_type * data() const;
```

5.162.4.1 ip::basic_endpoint::data (1 of 2 overloads)

Get the underlying endpoint in the native type.

```
data_type * data();
```

5.162.4.2 ip::basic_endpoint::data (2 of 2 overloads)

Get the underlying endpoint in the native type.

```
const data_type * data() const;
```

5.162.5 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

5.162.6 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

5.162.7 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

5.162.8 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.

5.162.9 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

5.162.10 ip::basic_endpoint::operator=

Assign from another endpoint.

```
basic_endpoint & operator=(
    const basic_endpoint & other);
```

Move-assign from another endpoint.

```
basic_endpoint & operator=(
    basic_endpoint && other);
```

5.162.10.1 ip::basic_endpoint::operator= (1 of 2 overloads)

Assign from another endpoint.

```
basic_endpoint & operator=(  
    const basic_endpoint & other);
```

5.162.10.2 ip::basic_endpoint::operator= (2 of 2 overloads)

Move-assign from another endpoint.

```
basic_endpoint & operator=(  
    basic_endpoint && other);
```

5.162.11 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

5.162.12 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

5.162.13 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

5.162.14 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);
```

5.162.14.1 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;
```

5.162.14.2 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);
```

5.162.15 ip::basic_endpoint::protocol

The protocol associated with the endpoint.

```
protocol_type protocol() const;
```

5.162.16 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

5.162.17 ip::basic_endpoint::resize

Set the underlying size of the endpoint in the native type.

```
void resize(  
    std::size_t new_size);
```

5.162.18 ip::basic_endpoint::size

Get the underlying size of the endpoint in the native type.

```
std::size_t size() const;
```

5.163 ip::basic_resolver

Provides endpoint resolution functionality.

```
template<
    typename InternetProtocol>
class basic_resolver :
    public ip::resolver_base
```

Types

Name	Description
endpoint_type	The endpoint type.
executor_type	The type of the executor associated with the object.
flags	A bitmask type (C++ Std [lib.bitmask.types]).
iterator	(Deprecated.) The iterator type.
protocol_type	The protocol type.
query	(Deprecated.) The query type.
results_type	The results type.

Member Functions

Name	Description
async_resolve	(Deprecated.) Asynchronously perform forward resolution of a query to a list of entries. 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. Move-construct a basic_resolver from another.
cancel	Cancel any asynchronous operations that are waiting on the resolver.
get_executor	Get the executor associated with the object.

Name	Description
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context associated with the object.
operator=	Move-assign a basic_resolver from another.
resolve	(Deprecated.) Perform forward resolution of a query to a list of entries. Perform forward resolution of a query to a list of entries. Perform reverse resolution of an endpoint to a list of entries.
~basic_resolver	Destroys the resolver.

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.
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` 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

5.163.1 ip::basic_resolver::address_configured

Inherited from ip::resolver_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;
```

5.163.2 ip::basic_resolver::all_matching

Inherited from ip::resolver_base.

If used with v4_mapped, return all matching IPv6 and IPv4 addresses.

```
static const flags all_matching = implementation_defined;
```

5.163.3 ip::basic_resolver::async_resolve

(Deprecated.) Asynchronously perform forward resolution of a query to a list of entries.

```
template<
    typename ResolveHandler>
DEDUCED async_resolve(
    const query & q,
    ResolveHandler && handler);
```

Asynchronously perform forward resolution of a query to a list of entries.

```
template<
    typename ResolveHandler>
DEDUCED async_resolve(
    string_view host,
    string_view service,
    ResolveHandler && handler);

template<
    typename ResolveHandler>
DEDUCED async_resolve(
    string_view host,
    string_view service,
    resolver_base::flags resolve_flags,
    ResolveHandler && handler);

template<
    typename ResolveHandler>
DEDUCED async_resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service,
```

```

ResolveHandler && handler);

template<
    typename ResolveHandler>
DEDUCED async_resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service,
    resolver_base::flags resolve_flags,
    ResolveHandler && handler);

```

Asynchronously perform reverse resolution of an endpoint to a list of entries.

```

template<
    typename ResolveHandler>
DEDUCED async_resolve(
    const endpoint_type & e,
    ResolveHandler && handler);

```

5.163.3.1 ip::basic_resolver::async_resolve (1 of 6 overloads)

(Deprecated.) Asynchronously perform forward resolution of a query to a list of entries.

```

template<
    typename ResolveHandler>
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::results_type results // Resolved endpoints as a range.
);

```

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_context::post()`.

A successful resolve operation is guaranteed to pass a non-empty range to the handler.

5.163.3.2 ip::basic_resolver::async_resolve (2 of 6 overloads)

Asynchronously perform forward resolution of a query to a list of entries.

```

template<
    typename ResolveHandler>
DEDUCED async_resolve(
    string_view host,
    string_view service,
    ResolveHandler && handler);

```

This function is used to resolve host and service names into a list of endpoint entries.

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.

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::results_type results // Resolved endpoints as a range.  
) ;
```

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_context::post()`.

A successful resolve operation is guaranteed to pass a non-empty range to the handler.

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.

5.163.3.3 ip::basic_resolver::async_resolve (3 of 6 overloads)

Asynchronously perform forward resolution of a query to a list of entries.

```
template<  
    typename ResolveHandler>  
DEDUCED async_resolve(  
    string_view host,  
    string_view service,  
    resolver_base::flags resolve_flags,  
    ResolveHandler && handler);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

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::results_type results // Resolved endpoints as a range.
);
```

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_context::post()`.

A successful resolve operation is guaranteed to pass a non-empty range to the handler.

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.

5.163.3.4 `ip::basic_resolver::async_resolve` (4 of 6 overloads)

Asynchronously perform forward resolution of a query to a list of entries.

```
template<
    typename ResolveHandler>
DEDUCED async_resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service,
    ResolveHandler && handler);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

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::results_type results // Resolved endpoints as a range.
);
```

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_context::post()`.

A successful resolve operation is guaranteed to pass a non-empty range to the handler.

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.

5.163.3.5 `ip::basic_resolver::async_resolve (5 of 6 overloads)`

Asynchronously perform forward resolution of a query to a list of entries.

```
template<
    typename ResolveHandler>
DEDUCED async_resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service,
    resolver_base::flags resolve_flags,
    ResolveHandler && handler);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

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::results_type results // Resolved endpoints as a range.
);
```

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_context::post()`.

A successful resolve operation is guaranteed to pass a non-empty range to the handler.

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.

5.163.3.6 ip::basic_resolver::async_resolve (6 of 6 overloads)

Asynchronously perform reverse resolution of an endpoint to a list of entries.

```
template<
    typename ResolveHandler>
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::results_type results // Resolved endpoints as a range.
);
```

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_context::post()`.

A successful resolve operation is guaranteed to pass a non-empty range to the handler.

5.163.4 ip::basic_resolver::basic_resolver

Constructor.

```
explicit basic_resolver(
    asio::io_context & io_context);
```

Move-construct a `ip::basic_resolver` from another.

```
basic_resolver(
    basic_resolver && other);
```

5.163.4.1 ip::basic_resolver::basic_resolver (1 of 2 overloads)

Constructor.

```
basic_resolver(
    asio::io_context & io_context);
```

This constructor creates a `ip::basic_resolver`.

Parameters

io_context The `io_context` object that the resolver will use to dispatch handlers for any asynchronous operations performed on the resolver.

5.163.4.2 ip::basic_resolver::basic_resolver (2 of 2 overloads)

Move-construct a `ip::basic_resolver` from another.

```
basic_resolver(
    basic_resolver && other);
```

This constructor moves a resolver from one object to another.

Parameters

other The other `ip::basic_resolver` 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_resolver(io_context&)` constructor.

5.163.5 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.

5.163.6 ip::basic_resolver::canonical_name

Inherited from ip::resolver_base.

Determine the canonical name of the host specified in the query.

```
static const flags canonical_name = implementation_defined;
```

5.163.7 ip::basic_resolver::endpoint_type

The endpoint type.

```
typedef InternetProtocol::endpoint endpoint_type;
```

Requirements

Header: `asio/ip/basic_resolver.hpp`

Convenience header: `asio.hpp`

5.163.8 ip::basic_resolver::executor_type

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/ip/basic_resolver.hpp

Convenience header: asio.hpp

5.163.9 ip::basic_resolver::flags

Inherited from ip::resolver_base.

A bitmask type (C++ Std [lib.bitmask.types]).

```
typedef unspecified flags;
```

Requirements

Header: asio/ip/basic_resolver.hpp

Convenience header: asio.hpp

5.163.10 ip::basic_resolver::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.163.11 ip::basic_resolver::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.163.12 ip::basic_resolver::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.163.13 ip::basic_resolver::iterator

(Deprecated.) 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
basic_resolver_iterator	Default constructor creates an end iterator. Copy constructor. Move constructor.
operator *	Dereference an iterator.
operator++	Increment operator (prefix). Increment operator (postfix).
operator->	Dereference an iterator.
operator=	Assignment operator. Move-assignment operator.

Protected Member Functions

Name	Description
dereference	
equal	
increment	

Protected Data Members

Name	Description
index_	
values_	

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

5.163.14 ip::basic_resolver::numeric_host

Inherited from ip::resolver_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;
```

5.163.15 ip::basic_resolver::numeric_service

Inherited from ip::resolver_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;
```

5.163.16 ip::basic_resolver::operator=

Move-assign a `ip::basic_resolver` from another.

```
basic_resolver & operator=(  
    basic_resolver && other);
```

This assignment operator moves a resolver from one object to another. Cancels any outstanding asynchronous operations associated with the target object.

Parameters

other The other `ip::basic_resolver` 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_resolver(io_context&)` constructor.

5.163.17 ip::basic_resolver::passive

Inherited from ip::resolver_base.

Indicate that returned endpoint is intended for use as a locally bound socket endpoint.

```
static const flags passive = implementation_defined;
```

5.163.18 ip::basic_resolver::protocol_type

The protocol type.

```
typedef InternetProtocol protocol_type;
```

Requirements

Header: asio/ip/basic_resolver.hpp

Convenience header: asio.hpp

5.163.19 ip::basic_resolver::query

(Deprecated.) 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.

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/basic_resolver.hpp`

Convenience header: `asio.hpp`

5.163.20 ip::basic_resolver::resolve

(Deprecated.) Perform forward resolution of a query to a list of entries.

```
results_type resolve(
    const query & q);

results_type resolve(
    const query & q,
    asio::error_code & ec);
```

Perform forward resolution of a query to a list of entries.

```
results_type resolve(
    string_view host,
    string_view service);

results_type resolve(
    string_view host,
    string_view service,
    asio::error_code & ec);

results_type resolve(
    string_view host,
```

```

        string_view service,
        resolver_base::flags resolve_flags);

results_type resolve(
    string_view host,
    string_view service,
    resolver_base::flags resolve_flags,
    asio::error_code & ec);

results_type resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service);
}

results_type resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service,
    asio::error_code & ec);

results_type resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service,
    resolver_base::flags resolve_flags);

results_type resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service,
    resolver_base::flags resolve_flags,
    asio::error_code & ec);

```

Perform reverse resolution of an endpoint to a list of entries.

```

results_type resolve(
    const endpoint_type & e);

results_type resolve(
    const endpoint_type & e,
    asio::error_code & ec);

```

5.163.20.1 ip::basic_resolver::resolve (1 of 12 overloads)

(Deprecated.) Perform forward resolution of a query to a list of entries.

```

results_type 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 range object representing the list of endpoint entries. A successful call to this function is guaranteed to return a non-empty range.

Exceptions

`asio::system_error` Thrown on failure.

5.163.20.2 `ip::basic_resolver::resolve (2 of 12 overloads)`

(Deprecated.) Perform forward resolution of a query to a list of entries.

```
results_type 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 range object representing the list of endpoint entries. An empty range is returned if an error occurs. A successful call to this function is guaranteed to return a non-empty range.

5.163.20.3 `ip::basic_resolver::resolve (3 of 12 overloads)`

Perform forward resolution of a query to a list of entries.

```
results_type resolve(
    string_view host,
    string_view service);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

Return Value

A range object representing the list of endpoint entries. A successful call to this function is guaranteed to return a non-empty range.

Exceptions

asio::system_error Thrown on failure.

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.

5.163.20.4 ip::basic_resolver::resolve (4 of 12 overloads)

Perform forward resolution of a query to a list of entries.

```
results_type resolve(
    string_view host,
    string_view service,
    asio::error_code & ec);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

ec Set to indicate what error occurred, if any.

Return Value

A range object representing the list of endpoint entries. An empty range is returned if an error occurs. A successful call to this function is guaranteed to return a non-empty range.

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.

5.163.20.5 ip::basic_resolver::resolve (5 of 12 overloads)

Perform forward resolution of a query to a list of entries.

```
results_type resolve(
    string_view host,
    string_view service,
    resolver_base::flags resolve_flags);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

Return Value

A range object representing the list of endpoint entries. A successful call to this function is guaranteed to return a non-empty range.

Exceptions

asio::system_error Thrown on failure.

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.

5.163.20.6 ip::basic_resolver::resolve (6 of 12 overloads)

Perform forward resolution of a query to a list of entries.

```
results_type resolve(
    string_view host,
    string_view service,
    resolver_base::flags resolve_flags,
    asio::error_code & ec);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

ec Set to indicate what error occurred, if any.

Return Value

A range object representing the list of endpoint entries. An empty range is returned if an error occurs. A successful call to this function is guaranteed to return a non-empty range.

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.

5.163.20.7 ip::basic_resolver::resolve (7 of 12 overloads)

Perform forward resolution of a query to a list of entries.

```
results_type resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

Return Value

A range object representing the list of endpoint entries. A successful call to this function is guaranteed to return a non-empty range.

Exceptions

asio::system_error Thrown on failure.

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.

5.163.20.8 ip::basic_resolver::resolve (8 of 12 overloads)

Perform forward resolution of a query to a list of entries.

```
results_type resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service,
    asio::error_code & ec);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

ec Set to indicate what error occurred, if any.

Return Value

A range object representing the list of endpoint entries. An empty range is returned if an error occurs. A successful call to this function is guaranteed to return a non-empty range.

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.

5.163.20.9 ip::basic_resolver::resolve (9 of 12 overloads)

Perform forward resolution of a query to a list of entries.

```
results_type resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service,
    resolver_base::flags resolve_flags);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

Return Value

A range object representing the list of endpoint entries. A successful call to this function is guaranteed to return a non-empty range.

Exceptions

asio::system_error Thrown on failure.

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.

5.163.20.10 ip::basic_resolver::resolve (10 of 12 overloads)

Perform forward resolution of a query to a list of entries.

```
results_type resolve(
    const protocol_type & protocol,
    string_view host,
    string_view service,
    resolver_base::flags resolve_flags,
    asio::error_code & ec);
```

This function is used to resolve host and service names into a list of endpoint entries.

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.

ec Set to indicate what error occurred, if any.

Return Value

A range object representing the list of endpoint entries. An empty range is returned if an error occurs. A successful call to this function is guaranteed to return a non-empty range.

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.

5.163.20.11 ip::basic_resolver::resolve (11 of 12 overloads)

Perform reverse resolution of an endpoint to a list of entries.

```
results_type 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 range object representing the list of endpoint entries. A successful call to this function is guaranteed to return a non-empty range.

Exceptions

asio::system_error Thrown on failure.

5.163.20.12 ip::basic_resolver::resolve (12 of 12 overloads)

Perform reverse resolution of an endpoint to a list of entries.

```
results_type 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 range object representing the list of endpoint entries. An empty range is returned if an error occurs. A successful call to this function is guaranteed to return a non-empty range.

5.163.21 ip::basic_resolver::results_type

The results type.

```
typedef basic_resolver_results< InternetProtocol > results_type;
```

Types

Name	Description
const_iterator	The type of an iterator into the range.
const_reference	The type of a const reference to a value in the range.
difference_type	Type used to represent the distance between two iterators in the range.
endpoint_type	The endpoint type associated with the results.
iterator	The type of an iterator into the range.
iterator_category	The iterator category.
pointer	The type of the result of applying operator->() to the iterator.
protocol_type	The protocol type associated with the results.
reference	The type of a non-const reference to a value in the range.
size_type	Type used to represent a count of the elements in the range.
value_type	The type of a value in the results range.

Member Functions

Name	Description
basic_resolver_results	Default constructor creates an empty range. Copy constructor. Move constructor.
begin	Obtain a begin iterator for the results range.
cbegin	Obtain a begin iterator for the results range.
cend	Obtain an end iterator for the results range.
empty	Determine whether the results range is empty.
end	Obtain an end iterator for the results range.
max_size	Get the maximum number of entries permitted in a results range.
operator *	Dereference an iterator.
operator++	Increment operator (prefix). Increment operator (postfix).
operator->	Dereference an iterator.
operator=	Assignment operator. Move-assignment operator.
size	Get the number of entries in the results range.
swap	Swap the results range with another.

Protected Member Functions

Name	Description
dereference	
equal	
increment	

Protected Data Members

Name	Description
index_	
values_	

Friends

Name	Description
operator!=	Test two iterators for inequality.
operator==	Test two iterators for equality.

The `ip::basic_resolver_results` class template is used to define a range over the results returned by a resolver.

The iterator's `value_type`, obtained when a results iterator is dereferenced, is:

```
const basic_resolver_entry<InternetProtocol>
```

Remarks

For backward compatibility, `ip::basic_resolver_results` is derived from `ip::basic_resolver_iterator`. This derivation is deprecated.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/basic_resolver.hpp`

Convenience header: `asio.hpp`

5.163.22 ip::basic_resolver::v4_mapped

Inherited from ip::resolver_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;
```

5.163.23 ip::basic_resolver::~basic_resolver

Destroys the resolver.

```
~basic_resolver();
```

This function destroys the resolver, cancelling any outstanding asynchronous wait operations associated with the resolver as if by calling `cancel`.

5.164 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`

5.164.1 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,
    string_view host,
    string_view service);
```

5.164.1.1 ip::basic_resolver_entry::basic_resolver_entry (1 of 2 overloads)

Default constructor.

```
basic_resolver_entry();
```

5.164.1.2 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,
    string_view host,
    string_view service);
```

5.164.2 ip::basic_resolver_entry::endpoint

Get the endpoint associated with the entry.

```
endpoint_type endpoint() const;
```

5.164.3 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

5.164.4 ip::basic_resolver_entry::host_name

Get the host name associated with the entry.

```
std::string host_name() const;

template<
    class Allocator>
std::basic_string< char, std::char_traits< char >, Allocator > host_name(
    const Allocator & alloc = Allocator()) const;
```

5.164.4.1 ip::basic_resolver_entry::host_name (1 of 2 overloads)

Get the host name associated with the entry.

```
std::string host_name() const;
```

5.164.4.2 ip::basic_resolver_entry::host_name (2 of 2 overloads)

Get the host name associated with the entry.

```
template<
    class Allocator>
std::basic_string< char, std::char_traits< char >, Allocator > host_name(
    const Allocator & alloc = Allocator()) const;
```

5.164.5 ip::basic_resolver_entry::operator endpoint_type

Convert to the endpoint associated with the entry.

```
operator endpoint_type() const;
```

5.164.6 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

5.164.7 ip::basic_resolver_entry::service_name

Get the service name associated with the entry.

```
std::string service_name() const;

template<
    class Allocator>
std::basic_string< char, std::char_traits< char >, Allocator > service_name(
    const Allocator & alloc = Allocator()) const;
```

5.164.7.1 ip::basic_resolver_entry::service_name (1 of 2 overloads)

Get the service name associated with the entry.

```
std::string service_name() const;
```

5.164.7.2 ip::basic_resolver_entry::service_name (2 of 2 overloads)

Get the service name associated with the entry.

```
template<
    class Allocator>
std::basic_string< char, std::char_traits< char >, Allocator > service_name(
    const Allocator & alloc = Allocator()) const;
```

5.165 ip::basic_resolver_iterator

An iterator over the entries produced by a resolver.

```
template<
    typename InternetProtocol>
class basic_resolver_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
basic_resolver_iterator	Default constructor creates an end iterator. Copy constructor. Move constructor.
operator *	Dereference an iterator.
operator++	Increment operator (prefix). Increment operator (postfix).
operator->	Dereference an iterator.
operator=	Assignment operator. Move-assignment operator.

Protected Member Functions

Name	Description
dereference	
equal	
increment	

Protected Data Members

Name	Description
index_	
values_	

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`

5.165.1 ip::basic_resolver_iterator::basic_resolver_iterator

Default constructor creates an end iterator.

```
basic_resolver_iterator();
```

Copy constructor.

```
basic_resolver_iterator(
    const basic_resolver_iterator & other);
```

Move constructor.

```
basic_resolver_iterator(
    basic_resolver_iterator && other);
```

5.165.1.1 ip::basic_resolver_iterator::basic_resolver_iterator (1 of 3 overloads)

Default constructor creates an end iterator.

```
basic_resolver_iterator();
```

5.165.1.2 ip::basic_resolver_iterator::basic_resolver_iterator (2 of 3 overloads)

Copy constructor.

```
basic_resolver_iterator(
    const basic_resolver_iterator & other);
```

5.165.1.3 ip::basic_resolver_iterator::basic_resolver_iterator (3 of 3 overloads)

Move constructor.

```
basic_resolver_iterator(
    basic_resolver_iterator && other);
```

5.165.2 ip::basic_resolver_iterator::dereference

```
const basic_resolver_entry< InternetProtocol > & dereference() const;
```

5.165.3 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

5.165.4 ip::basic_resolver_iterator::equal

```
bool equal(
    const basic_resolver_iterator & other) const;
```

5.165.5 ip::basic_resolver_iterator::increment

```
void increment();
```

5.165.6 ip::basic_resolver_iterator::index_

```
std::size_t index_;
```

5.165.7 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

5.165.8 ip::basic_resolver_iterator::operator *

Dereference an iterator.

```
const basic_resolver_entry< InternetProtocol > & operator *() const;
```

5.165.9 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

5.165.10 ip::basic_resolver_iterator::operator++

Increment operator (prefix).

```
basic_resolver_iterator & operator++();
```

Increment operator (postfix).

```
basic_resolver_iterator operator++(
    int );
```

5.165.10.1 ip::basic_resolver_iterator::operator++ (1 of 2 overloads)

Increment operator (prefix).

```
basic_resolver_iterator & operator++();
```

5.165.10.2 ip::basic_resolver_iterator::operator++ (2 of 2 overloads)

Increment operator (postfix).

```
basic_resolver_iterator operator++(
    int );
```

5.165.11 ip::basic_resolver_iterator::operator->

Dereference an iterator.

```
const basic_resolver_entry< InternetProtocol > * operator->() const;
```

5.165.12 ip::basic_resolver_iterator::operator=

Assignment operator.

```
basic_resolver_iterator & operator=(  
    const basic_resolver_iterator & other);
```

Move-assignment operator.

```
basic_resolver_iterator & operator=(  
    basic_resolver_iterator && other);
```

5.165.12.1 ip::basic_resolver_iterator::operator= (1 of 2 overloads)

Assignment operator.

```
basic_resolver_iterator & operator=(  
    const basic_resolver_iterator & other);
```

5.165.12.2 ip::basic_resolver_iterator::operator= (2 of 2 overloads)

Move-assignment operator.

```
basic_resolver_iterator & operator=(  
    basic_resolver_iterator && other);
```

5.165.13 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

5.165.14 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

5.165.15 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

5.165.16 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`

5.165.17 ip::basic_resolver_iterator::values_

```
values_ptr_type values_;
```

5.166 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.
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.

Name	Description
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`

5.166.1 ip::basic_resolver_query::address_configured

Inherited from ip::resolver_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;
```

5.166.2 ip::basic_resolver_query::all_matching

Inherited from ip::resolver_base.

If used with v4_mapped, return all matching IPv6 and IPv4 addresses.

```
static const flags all_matching = implementation_defined;
```

5.166.3 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);
```

5.166.3.1 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.

5.166.3.2 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.

5.166.3.3 `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.

5.166.3.4 `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.

5.166.4 ip::basic_resolver_query::canonical_name

Inherited from ip::resolver_base.

Determine the canonical name of the host specified in the query.

```
static const flags canonical_name = implementation_defined;
```

5.166.5 ip::basic_resolver_query::flags

Inherited from ip::resolver_base.

A bitmask type (C++ Std [lib.bitmask.types]).

```
typedef unspecified flags;
```

Requirements

Header: asio/ip/basic_resolver_query.hpp

Convenience header: asio.hpp

5.166.6 ip::basic_resolver_query::hints

Get the hints associated with the query.

```
const asio::detail::addrinfo_type & hints() const;
```

5.166.7 ip::basic_resolver_query::host_name

Get the host name associated with the query.

```
std::string host_name() const;
```

5.166.8 ip::basic_resolver_query::numeric_host

Inherited from ip::resolver_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;
```

5.166.9 ip::basic_resolver_query::numeric_service

Inherited from ip::resolver_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;
```

5.166.10 ip::basic_resolver_query::passive

Inherited from ip::resolver_base.

Indicate that returned endpoint is intended for use as a locally bound socket endpoint.

```
static const flags passive = implementation_defined;
```

5.166.11 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

5.166.12 ip::basic_resolver_query::service_name

Get the service name associated with the query.

```
std::string service_name() const;
```

5.166.13 ip::basic_resolver_query::v4_mapped

Inherited from ip::resolver_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;
```

5.167 ip::basic_resolver_results

A range of entries produced by a resolver.

```
template<
    typename InternetProtocol>
class basic_resolver_results :
    public ip::basic_resolver_iterator< InternetProtocol >
```

Types

Name	Description
const_iterator	The type of an iterator into the range.
const_reference	The type of a const reference to a value in the range.
difference_type	Type used to represent the distance between two iterators in the range.
endpoint_type	The endpoint type associated with the results.
iterator	The type of an iterator into the range.
iterator_category	The iterator category.
pointer	The type of the result of applying operator->() to the iterator.
protocol_type	The protocol type associated with the results.
reference	The type of a non-const reference to a value in the range.
size_type	Type used to represent a count of the elements in the range.
value_type	The type of a value in the results range.

Member Functions

Name	Description
basic_resolver_results	Default constructor creates an empty range. Copy constructor. Move constructor.
begin	Obtain a begin iterator for the results range.
cbegin	Obtain a begin iterator for the results range.
cend	Obtain an end iterator for the results range.
empty	Determine whether the results range is empty.
end	Obtain an end iterator for the results range.

Name	Description
max_size	Get the maximum number of entries permitted in a results range.
operator *	Dereference an iterator.
operator++	Increment operator (prefix). Increment operator (postfix).
operator->	Dereference an iterator.
operator=	Assignment operator. Move-assignment operator.
size	Get the number of entries in the results range.
swap	Swap the results range with another.

Protected Member Functions

Name	Description
dereference	
equal	
increment	

Protected Data Members

Name	Description
index_	
values_	

Friends

Name	Description
operator!=	Test two iterators for inequality.
operator==	Test two iterators for equality.

The `ip::basic_resolver_results` class template is used to define a range over the results returned by a resolver.

The iterator's value_type, obtained when a results iterator is dereferenced, is:

```
const basic_resolver_entry<InternetProtocol>
```

Remarks

For backward compatibility, `ip::basic_resolver_results` is derived from `ip::basic_resolver_iterator`. This derivation is deprecated.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/basic_resolver_results.hpp

Convenience header: asio.hpp

5.167.1 ip::basic_resolver_results::basic_resolver_results

Default constructor creates an empty range.

```
basic_resolver_results();
```

Copy constructor.

```
basic_resolver_results(  
    const basic_resolver_results & other);
```

Move constructor.

```
basic_resolver_results(  
    basic_resolver_results && other);
```

5.167.1.1 ip::basic_resolver_results::basic_resolver_results (1 of 3 overloads)

Default constructor creates an empty range.

```
basic_resolver_results();
```

5.167.1.2 ip::basic_resolver_results::basic_resolver_results (2 of 3 overloads)

Copy constructor.

```
basic_resolver_results(  
    const basic_resolver_results & other);
```

5.167.1.3 ip::basic_resolver_results::basic_resolver_results (3 of 3 overloads)

Move constructor.

```
basic_resolver_results(  
    basic_resolver_results && other);
```

5.167.2 ip::basic_resolver_results::begin

Obtain a begin iterator for the results range.

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

5.167.3 ip::basic_resolver_results::cbegin

Obtain a begin iterator for the results range.

```
const_iterator cbegin() const;
```

5.167.4 ip::basic_resolver_results::cend

Obtain an end iterator for the results range.

```
const_iterator cend() const;
```

5.167.5 ip::basic_resolver_results::const_iterator

The type of an iterator into the range.

```
typedef basic_resolver_iterator< protocol_type > const_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
basic_resolver_iterator	Default constructor creates an end iterator. Copy constructor. Move constructor.
operator *	Dereference an iterator.
operator++	Increment operator (prefix). Increment operator (postfix).
operator->	Dereference an iterator.
operator=	Assignment operator. Move-assignment operator.

Protected Member Functions

Name	Description
dereference	
equal	
increment	

Protected Data Members

Name	Description
index_	
values_	

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_results.hpp`

Convenience header: `asio.hpp`

5.167.6 ip::basic_resolver_results::const_reference

The type of a const reference to a value in the range.

```
typedef const value_type & const_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_results.hpp`

Convenience header: `asio.hpp`

5.167.7 ip::basic_resolver_results::dereference

Inherited from `ip::basic_resolver`.

```
const basic_resolver_entry< InternetProtocol > & dereference() const;
```

5.167.8 ip::basic_resolver_results::difference_type

Type used to represent the distance between two iterators in the range.

```
typedef std::ptrdiff_t difference_type;
```

Requirements

Header: `asio/ip/basic_resolver_results.hpp`

Convenience header: `asio.hpp`

5.167.9 ip::basic_resolver_results::empty

Determine whether the results range is empty.

```
bool empty() const;
```

5.167.10 ip::basic_resolver_results::end

Obtain an end iterator for the results range.

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

5.167.11 ip::basic_resolver_results::endpoint_type

The endpoint type associated with the results.

```
typedef protocol_type::endpoint endpoint_type;
```

Requirements

Header: asio/ip/basic_resolver_results.hpp

Convenience header: asio.hpp

5.167.12 ip::basic_resolver_results::equal

Inherited from ip::basic_resolver.

```
bool equal(
    const basic_resolver_iterator & other) const;
```

5.167.13 ip::basic_resolver_results::increment

Inherited from ip::basic_resolver.

```
void increment();
```

5.167.14 ip::basic_resolver_results::index_

Inherited from ip::basic_resolver.

```
std::size_t index_;
```

5.167.15 ip::basic_resolver_results::iterator

The type of an iterator into the range.

```
typedef const_iterator 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
basic_resolver_iterator	Default constructor creates an end iterator. Copy constructor. Move constructor.
operator *	Dereference an iterator.
operator++	Increment operator (prefix). Increment operator (postfix).
operator->	Dereference an iterator.
operator=	Assignment operator. Move-assignment operator.

Protected Member Functions

Name	Description
dereference	
equal	
increment	

Protected Data Members

Name	Description
index_	
values_	

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_results.hpp`

Convenience header: `asio.hpp`

5.167.16 ip::basic_resolver_results::iterator_category

Inherited from ip::basic_resolver:

The iterator category.

```
typedef std::forward_iterator_tag iterator_category;
```

Requirements

Header: `asio/ip/basic_resolver_results.hpp`

Convenience header: `asio.hpp`

5.167.17 ip::basic_resolver_results::max_size

Get the maximum number of entries permitted in a results range.

```
size_type max_size() const;
```

5.167.18 ip::basic_resolver_results::operator *

Inherited from ip::basic_resolver:

Dereference an iterator.

```
const basic_resolver_entry< InternetProtocol > & operator *() const;
```

5.167.19 ip::basic_resolver_results::operator!=

Test two iterators for inequality.

```
friend bool operator!=(
    const basic_resolver_results & a,
    const basic_resolver_results & b);

friend bool operator!=(
    const basic_resolver_iterator & a,
    const basic_resolver_iterator & b);
```

5.167.19.1 ip::basic_resolver_results::operator!= (1 of 2 overloads)

Test two iterators for inequality.

```
friend bool operator!=(
    const basic_resolver_results & a,
    const basic_resolver_results & b);
```

Requirements

Header: asio/ip/basic_resolver_results.hpp

Convenience header: asio.hpp

5.167.19.2 ip::basic_resolver_results::operator!= (2 of 2 overloads)

Inherited from ip::basic_resolver.

Test two iterators for inequality.

```
friend bool operator!=(
    const basic_resolver_iterator & a,
    const basic_resolver_iterator & b);
```

Requirements

Header: asio/ip/basic_resolver_results.hpp

Convenience header: asio.hpp

5.167.20 ip::basic_resolver_results::operator++

Increment operator (prefix).

```
basic_resolver_iterator & operator++();
```

Increment operator (postfix).

```
basic_resolver_iterator operator++(
    int );
```

5.167.20.1 ip::basic_resolver_results::operator++ (1 of 2 overloads)

Inherited from ip::basic_resolver.

Increment operator (prefix).

```
basic_resolver_iterator & operator++();
```

5.167.20.2 ip::basic_resolver_results::operator++ (2 of 2 overloads)

Inherited from ip::basic_resolver.

Increment operator (postfix).

```
basic_resolver_iterator operator++(
    int );
```

5.167.21 ip::basic_resolver_results::operator->

Inherited from ip::basic_resolver.

Dereference an iterator.

```
const basic_resolver_entry< InternetProtocol > * operator->() const;
```

5.167.22 ip::basic_resolver_results::operator=

Assignment operator.

```
basic_resolver_results & operator=(
    const basic_resolver_results & other);
```

Move-assignment operator.

```
basic_resolver_results & operator=(
    basic_resolver_results && other);
```

5.167.22.1 ip::basic_resolver_results::operator=(1 of 2 overloads)

Assignment operator.

```
basic_resolver_results & operator=(
    const basic_resolver_results & other);
```

5.167.22.2 ip::basic_resolver_results::operator=(2 of 2 overloads)

Move-assignment operator.

```
basic_resolver_results & operator=(
    basic_resolver_results && other);
```

5.167.23 ip::basic_resolver_results::operator==

Test two iterators for equality.

```
friend bool operator==(  
    const basic_resolver_results & a,  
    const basic_resolver_results & b);  
  
friend bool operator==(  
    const basic_resolver_iterator & a,  
    const basic_resolver_iterator & b);
```

5.167.23.1 ip::basic_resolver_results::operator== (1 of 2 overloads)

Test two iterators for equality.

```
friend bool operator==(  
    const basic_resolver_results & a,  
    const basic_resolver_results & b);
```

Requirements

Header: asio/ip/basic_resolver_results.hpp

Convenience header: asio.hpp

5.167.23.2 ip::basic_resolver_results::operator== (2 of 2 overloads)

Inherited from ip::basic_resolver.

Test two iterators for equality.

```
friend bool operator==(  
    const basic_resolver_iterator & a,  
    const basic_resolver_iterator & b);
```

Requirements

Header: asio/ip/basic_resolver_results.hpp

Convenience header: asio.hpp

5.167.24 ip::basic_resolver_results::pointer

Inherited from ip::basic_resolver.

The type of the result of applying operator->() to the iterator.

```
typedef const basic_resolver_entry< InternetProtocol > * pointer;
```

Requirements

Header: asio/ip/basic_resolver_results.hpp

Convenience header: asio.hpp

5.167.25 ip::basic_resolver_results::protocol_type

The protocol type associated with the results.

```
typedef InternetProtocol protocol_type;
```

Requirements

Header: asio/ip/basic_resolver_results.hpp

Convenience header: asio.hpp

5.167.26 ip::basic_resolver_results::reference

The type of a non-const reference to a value in the range.

```
typedef value_type & 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_results.hpp

Convenience header: asio.hpp

5.167.27 ip::basic_resolver_results::size

Get the number of entries in the results range.

```
size_type size() const;
```

5.167.28 ip::basic_resolver_results::size_type

Type used to represent a count of the elements in the range.

```
typedef std::size_t size_type;
```

Requirements

Header: asio/ip/basic_resolver_results.hpp

Convenience header: asio.hpp

5.167.29 ip::basic_resolver_results::swap

Swap the results range with another.

```
void swap(
    basic_resolver_results & that);
```

5.167.30 ip::basic_resolver_results::value_type

The type of a value in the results range.

```
typedef basic_resolver_entry< endpoint_type > 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_results.hpp`

Convenience header: `asio.hpp`

5.167.31 ip::basic_resolver_results::values_

Inherited from ip::basic_resolver.

```
values_ptr_type values_;
```

5.168 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`

5.168.1 ip::host_name (1 of 2 overloads)

Get the current host name.

```
std::string host_name();
```

5.168.2 ip::host_name (2 of 2 overloads)

Get the current host name.

```
std::string host_name(  
    asio::error_code & ec);
```

5.169 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.

Name	Description
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`

5.169.1 ip::icmp::endpoint

The type of a ICMP endpoint.

```
typedef basic_endpoint< icmp > 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>address</code>	Get the IP address associated with the endpoint. Set the IP address associated with the endpoint.
<code>basic_endpoint</code>	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. Move constructor.

Name	Description
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. Move-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

5.169.2 ip::icmp::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.169.3 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

5.169.4 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

5.169.5 ip::icmp::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.169.6 ip::icmp::resolver

The ICMP resolver type.

```
typedef basic_resolver< icmp > resolver;
```

Types

Name	Description
endpoint_type	The endpoint type.
executor_type	The type of the executor associated with the object.
flags	A bitmask type (C++ Std [lib.bitmask.types]).
iterator	(Deprecated.) The iterator type.
protocol_type	The protocol type.
query	(Deprecated.) The query type.
results_type	The results type.

Member Functions

Name	Description
async_resolve	(Deprecated.) Asynchronously perform forward resolution of a query to a list of entries. 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. Move-construct a basic_resolver from another.
cancel	Cancel any asynchronous operations that are waiting on the resolver.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context associated with the object.
operator=	Move-assign a basic_resolver from another.
resolve	(Deprecated.) Perform forward resolution of a query to a list of entries. Perform forward resolution of a query to a list of entries. Perform reverse resolution of an endpoint to a list of entries.
~basic_resolver	Destroys the resolver.

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.
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` 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`

5.169.7 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.

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.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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_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.
release	Release ownership of the underlying native socket.
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
~basic_raw_socket	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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

5.169.8 ip::icmp::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.169.9 ip::icmp::v4

Construct to represent the IPv4 ICMP protocol.

```
static icmp v4();
```

5.169.10 ip::icmp::v6

Construct to represent the IPv6 ICMP protocol.

```
static icmp v6();
```

5.170 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_context);
...
asio::ip::multicast::enable_loopback option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.171 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_context);
...
asio::ip::multicast::hops option(4);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
asio::ip::multicast::hops option;
socket.get_option(option);
int ttl = option.value();
```

Requirements

Header: asio/ip/multicast.hpp

Convenience header: asio.hpp

5.172 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_context);
...
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

5.173 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_context);
...
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

5.174 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_context);
...
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

5.175 ip::network_v4

Represents an IPv4 network.

```
class network_v4
```

Member Functions

Name	Description
address	Obtain the address object specified when the network object was created.
broadcast	Obtain an address object that represents the network's broadcast address.
canonical	Obtain the true network address, omitting any host bits.
hosts	Obtain an address range corresponding to the hosts in the network.
is_host	Test if network is a valid host address.
is_subnet_of	Test if a network is a real subnet of another network.
netmask	Obtain the netmask that was specified when the network object was created.
network	Obtain an address object that represents the network address.
network_v4	Default constructor. Construct a network based on the specified address and prefix length. Construct network based on the specified address and netmask. Copy constructor.
operator=	Assign from another network.
prefix_length	Obtain the prefix length that was specified when the network object was created.
to_string	Get the network as an address in dotted decimal format.

Friends

Name	Description
operator!=	Compare two networks for inequality.
operator==	Compare two networks for equality.

Related Functions

Name	Description
make_network_v4	Create an IPv4 network from a string containing IP address and prefix length.

The [ip::network_v4](#) class provides the ability to use and manipulate IP version 4 networks.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/ip/network_v4.hpp`

Convenience header: `asio.hpp`

5.175.1 ip::network_v4::address

Obtain the address object specified when the network object was created.

```
address_v4 address() const;
```

5.175.2 ip::network_v4::broadcast

Obtain an address object that represents the network's broadcast address.

```
address_v4 broadcast() const;
```

5.175.3 ip::network_v4::canonical

Obtain the true network address, omitting any host bits.

```
network_v4 canonical() const;
```

5.175.4 ip::network_v4::hosts

Obtain an address range corresponding to the hosts in the network.

```
address_v4_range hosts() const;
```

5.175.5 ip::network_v4::is_host

Test if network is a valid host address.

```
bool is_host() const;
```

5.175.6 ip::network_v4::is_subnet_of

Create an IPv4 network from a string containing IP address and prefix length.

```
bool is_subnet_of(
    const network_v4 & other) const;
```

5.175.7 ip::network_v4::make_network_v4

Create an IPv4 network from a string containing IP address and prefix length.

```
network_v4 make_network_v4(
    const char * str);

network_v4 make_network_v4(
    const char * str,
    asio::error_code & ec);

network_v4 make_network_v4(
    const std::string & str);

network_v4 make_network_v4(
    const std::string & str,
    asio::error_code & ec);

network_v4 make_network_v4(
    string_view str);

network_v4 make_network_v4(
    string_view str,
    asio::error_code & ec);
```

5.175.7.1 ip::network_v4::make_network_v4 (1 of 6 overloads)

Create an IPv4 network from a string containing IP address and prefix length.

```
network_v4 make_network_v4(
    const char * str);
```

5.175.7.2 ip::network_v4::make_network_v4 (2 of 6 overloads)

Create an IPv4 network from a string containing IP address and prefix length.

```
network_v4 make_network_v4(
    const char * str,
    asio::error_code & ec);
```

5.175.7.3 ip::network_v4::make_network_v4 (3 of 6 overloads)

Create an IPv4 network from a string containing IP address and prefix length.

```
network_v4 make_network_v4(
    const std::string & str);
```

5.175.7.4 ip::network_v4::make_network_v4 (4 of 6 overloads)

Create an IPv4 network from a string containing IP address and prefix length.

```
network_v4 make_network_v4(
    const std::string & str,
    asio::error_code & ec);
```

5.175.7.5 ip::network_v4::make_network_v4 (5 of 6 overloads)

Create an IPv4 network from a string containing IP address and prefix length.

```
network_v4 make_network_v4(
    string_view str);
```

5.175.7.6 ip::network_v4::make_network_v4 (6 of 6 overloads)

Create an IPv4 network from a string containing IP address and prefix length.

```
network_v4 make_network_v4(
    string_view str,
    asio::error_code & ec);
```

5.175.8 ip::network_v4::netmask

Obtain the netmask that was specified when the network object was created.

```
address_v4 netmask() const;
```

5.175.9 ip::network_v4::network

Obtain an address object that represents the network address.

```
address_v4 network() const;
```

5.175.10 ip::network_v4::network_v4

Default constructor.

```
network_v4();
```

Construct a network based on the specified address and prefix length.

```
network_v4(
    const address_v4 & addr,
    unsigned short prefix_len);
```

Construct network based on the specified address and netmask.

```
network_v4(
    const address_v4 & addr,
    const address_v4 & mask);
```

Copy constructor.

```
network_v4 (const network_v4 & other);
```

5.175.10.1 ip::network_v4::network_v4 (1 of 4 overloads)

Default constructor.

```
network_v4();
```

5.175.10.2 ip::network_v4::network_v4 (2 of 4 overloads)

Construct a network based on the specified address and prefix length.

```
network_v4 (const address_v4 & addr, unsigned short prefix_len);
```

5.175.10.3 ip::network_v4::network_v4 (3 of 4 overloads)

Construct network based on the specified address and netmask.

```
network_v4 (const address_v4 & addr, const address_v4 & mask);
```

5.175.10.4 ip::network_v4::network_v4 (4 of 4 overloads)

Copy constructor.

```
network_v4 (const network_v4 & other);
```

5.175.11 ip::network_v4::operator!=

Compare two networks for inequality.

```
friend bool operator!=(const network_v4 & a, const network_v4 & b);
```

Requirements

Header: asio/ip/network_v4.hpp

Convenience header: asio.hpp

5.175.12 ip::network_v4::operator=

Assign from another network.

```
network_v4 & operator=(const network_v4 & other);
```

5.175.13 ip::network_v4::operator==

Compare two networks for equality.

```
friend bool operator==
  const network_v4 & a,
  const network_v4 & b);
```

Requirements

Header: asio/ip/network_v4.hpp

Convenience header: asio.hpp

5.175.14 ip::network_v4::prefix_length

Obtain the prefix length that was specified when the network object was created.

```
unsigned short prefix_length() const;
```

5.175.15 ip::network_v4::to_string

Get the network as an address in dotted decimal format.

```
std::string to_string() const;

std::string to_string(
  asio::error_code & ec) const;
```

5.175.15.1 ip::network_v4::to_string (1 of 2 overloads)

Get the network as an address in dotted decimal format.

```
std::string to_string() const;
```

5.175.15.2 ip::network_v4::to_string (2 of 2 overloads)

Get the network as an address in dotted decimal format.

```
std::string to_string(
  asio::error_code & ec) const;
```

5.176 ip::network_v6

Represents an IPv6 network.

```
class network_v6
```

Member Functions

Name	Description
address	Obtain the address object specified when the network object was created.
canonical	Obtain the true network address, omitting any host bits.
hosts	Obtain an address range corresponding to the hosts in the network.
is_host	Test if network is a valid host address.
is_subnet_of	Test if a network is a real subnet of another network.
network	Obtain an address object that represents the network address.
network_v6	Default constructor. Construct a network based on the specified address and prefix length. Copy constructor.
operator=	Assign from another network.
prefix_length	Obtain the prefix length that was specified when the network object was created.
to_string	Get the network as an address in dotted decimal format.

Friends

Name	Description
operator!=	Compare two networks for inequality.
operator==	Compare two networks for equality.

Related Functions

Name	Description
make_network_v6	Create an IPv6 network from a string containing IP address and prefix length.

The `ip::network_v6` class provides the ability to use and manipulate IP version 6 networks.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/ip/network_v6.hpp

Convenience header: asio.hpp

5.176.1 ip::network_v6::address

Obtain the address object specified when the network object was created.

```
address_v6 address() const;
```

5.176.2 ip::network_v6::canonical

Obtain the true network address, omitting any host bits.

```
network_v6 canonical() const;
```

5.176.3 ip::network_v6::hosts

Obtain an address range corresponding to the hosts in the network.

```
address_v6_range hosts() const;
```

5.176.4 ip::network_v6::is_host

Test if network is a valid host address.

```
bool is_host() const;
```

5.176.5 ip::network_v6::is_subnet_of

Test if a network is a real subnet of another network.

```
bool is_subnet_of(
    const network_v6 & other) const;
```

5.176.6 ip::network_v6::make_network_v6

Create an IPv6 network from a string containing IP address and prefix length.

```
network_v6 make_network_v6(
    const char * str);
```

```
network_v6 make_network_v6(
    const char * str,
    asio::error_code & ec);
```

```
network_v6 make_network_v6(
    const std::string & str);
```

```
network_v6 make_network_v6(
    const std::string & str,
    asio::error_code & ec);

network_v6 make_network_v6(
    string_view str);

network_v6 make_network_v6(
    string_view str,
    asio::error_code & ec);
```

5.176.6.1 ip::network_v6::make_network_v6 (1 of 6 overloads)

Create an IPv6 network from a string containing IP address and prefix length.

```
network_v6 make_network_v6(
    const char * str);
```

5.176.6.2 ip::network_v6::make_network_v6 (2 of 6 overloads)

Create an IPv6 network from a string containing IP address and prefix length.

```
network_v6 make_network_v6(
    const char * str,
    asio::error_code & ec);
```

5.176.6.3 ip::network_v6::make_network_v6 (3 of 6 overloads)

Create an IPv6 network from a string containing IP address and prefix length.

```
network_v6 make_network_v6(
    const std::string & str);
```

5.176.6.4 ip::network_v6::make_network_v6 (4 of 6 overloads)

Create an IPv6 network from a string containing IP address and prefix length.

```
network_v6 make_network_v6(
    const std::string & str,
    asio::error_code & ec);
```

5.176.6.5 ip::network_v6::make_network_v6 (5 of 6 overloads)

Create an IPv6 network from a string containing IP address and prefix length.

```
network_v6 make_network_v6(
    string_view str);
```

5.176.6.6 ip::network_v6::make_network_v6 (6 of 6 overloads)

Create an IPv6 network from a string containing IP address and prefix length.

```
network_v6 make_network_v6(
    string_view str,
    asio::error_code & ec);
```

5.176.7 ip::network_v6::network

Obtain an address object that represents the network address.

```
address_v6 network() const;
```

5.176.8 ip::network_v6::network_v6

Default constructor.

```
network_v6();
```

Construct a network based on the specified address and prefix length.

```
network_v6(
    const address_v6 & addr,
    unsigned short prefix_len);
```

Copy constructor.

```
network_v6(
    const network_v6 & other);
```

5.176.8.1 ip::network_v6::network_v6 (1 of 3 overloads)

Default constructor.

```
network_v6();
```

5.176.8.2 ip::network_v6::network_v6 (2 of 3 overloads)

Construct a network based on the specified address and prefix length.

```
network_v6(
    const address_v6 & addr,
    unsigned short prefix_len);
```

5.176.8.3 ip::network_v6::network_v6 (3 of 3 overloads)

Copy constructor.

```
network_v6(
    const network_v6 & other);
```

5.176.9 ip::network_v6::operator!=

Compare two networks for inequality.

```
friend bool operator!=(
    const network_v6 & a,
    const network_v6 & b);
```

Requirements

Header: asio/ip/network_v6.hpp

Convenience header: asio.hpp

5.176.10 ip::network_v6::operator=

Assign from another network.

```
network_v6 & operator=(  
    const network_v6 & other);
```

5.176.11 ip::network_v6::operator==

Compare two networks for equality.

```
friend bool operator==(  
    const network_v6 & a,  
    const network_v6 & b);
```

Requirements

Header: asio/ip/network_v6.hpp

Convenience header: asio.hpp

5.176.12 ip::network_v6::prefix_length

Obtain the prefix length that was specified when the network object was created.

```
unsigned short prefix_length() const;
```

5.176.13 ip::network_v6::to_string

Get the network as an address in dotted decimal format.

```
std::string to_string() const;  
  
std::string to_string(  
    asio::error_code & ec) const;
```

5.176.13.1 ip::network_v6::to_string (1 of 2 overloads)

Get the network as an address in dotted decimal format.

```
std::string to_string() const;
```

5.176.13.2 ip::network_v6::to_string (2 of 2 overloads)

Get the network as an address in dotted decimal format.

```
std::string to_string(  
    asio::error_code & ec) const;
```

5.177 ip::resolver_base

The `ip::resolver_base` class is used as a base for the `ip::basic_resolver` class templates to provide a common place to define the flag constants.

```
class resolver_base
```

Types

Name	Description
<code>flags</code>	A bitmask type (C++ Std [lib.bitmask.types]).

Protected Member Functions

Name	Description
<code>~resolver_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.
<code>numeric_host</code>	Host name should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.
<code>numeric_service</code>	Service name should be treated as a numeric string defining a port number and no name resolution should be attempted.
<code>passive</code>	Indicate that returned endpoint is intended for use as a locally bound socket endpoint.
<code>v4_mapped</code>	If the query protocol family is specified as IPv6, return IPv4-mapped IPv6 addresses on finding no IPv6 addresses.

Requirements

Header: `asio/ip/resolver_base.hpp`

Convenience header: asio.hpp

5.177.1 ip::resolver_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;
```

5.177.2 ip::resolver_base::all_matching

If used with v4_mapped, return all matching IPv6 and IPv4 addresses.

```
static const flags all_matching = implementation_defined;
```

5.177.3 ip::resolver_base::canonical_name

Determine the canonical name of the host specified in the query.

```
static const flags canonical_name = implementation_defined;
```

5.177.4 ip::resolver_base::flags

A bitmask type (C++ Std [lib.bitmask.types]).

```
typedef unspecified flags;
```

Requirements

Header: asio/ip/resolver_base.hpp

Convenience header: asio.hpp

5.177.5 ip::resolver_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;
```

5.177.6 ip::resolver_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;
```

5.177.7 ip::resolver_base::passive

Indicate that returned endpoint is intended for use as a locally bound socket endpoint.

```
static const flags passive = implementation_defined;
```

5.177.8 ip::resolver_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;
```

5.177.9 ip::resolver_base::~resolver_base

Protected destructor to prevent deletion through this type.

```
~resolver_base();
```

5.178 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 :  
public ip::resolver_base
```

Types

Name	Description
flags	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.
<code>numeric_host</code>	Host name should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.

Name	Description
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

5.178.1 ip::resolver_query_base::address_configured

Inherited from ip::resolver_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;
```

5.178.2 ip::resolver_query_base::all_matching

Inherited from ip::resolver_base.

If used with v4_mapped, return all matching IPv6 and IPv4 addresses.

```
static const flags all_matching = implementation_defined;
```

5.178.3 ip::resolver_query_base::canonical_name

Inherited from ip::resolver_base.

Determine the canonical name of the host specified in the query.

```
static const flags canonical_name = implementation_defined;
```

5.178.4 ip::resolver_query_base::flags

Inherited from ip::resolver_base.

A bitmask type (C++ Std [lib.bitmask.types]).

```
typedef unspecified flags;
```

Requirements

Header: asio/ip/resolver_query_base.hpp

Convenience header: asio.hpp

5.178.5 ip::resolver_query_base::numeric_host

Inherited from ip::resolver_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;
```

5.178.6 ip::resolver_query_base::numeric_service

Inherited from ip::resolver_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;
```

5.178.7 ip::resolver_query_base::passive

Inherited from ip::resolver_base.

Indicate that returned endpoint is intended for use as a locally bound socket endpoint.

```
static const flags passive = implementation_defined;
```

5.178.8 ip::resolver_query_base::v4_mapped

Inherited from ip::resolver_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;
```

5.178.9 ip::resolver_query_base::~resolver_query_base

Protected destructor to prevent deletion through this type.

```
~resolver_query_base();
```

5.179 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.

Name	Description
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`

5.179.1 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.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.
async_wait	Asynchronously wait for the acceptor to become ready to read, ready to write, or to have pending error conditions.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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.

Name	Description
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.
release	Release ownership of the underlying native acceptor.
set_option	Set an option on the acceptor.
wait	Wait for the acceptor to become ready to read, ready to write, or to have pending error conditions.
~basic_socket_acceptor	Destroys the acceptor.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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_context);
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

5.179.2 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. Move 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. Move-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.

Name	Description
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/tcp.hpp`

Convenience header: `asio.hpp`

5.179.3 ip::tcp::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.179.4 ip::tcp::iostream

The TCP iostream type.

```
typedef basic_socket_iostream< tcp > iostream;
```

Types

Name	Description
clock_type	The clock type.
duration	The duration type.
duration_type	(Deprecated: Use duration.) The duration type.
endpoint_type	The endpoint type.
protocol_type	The protocol type.
time_point	The time type.
time_type	(Deprecated: Use time_point.) The time type.

Member Functions

Name	Description
basic_socket_iostream	Construct a basic_socket_iostream without establishing a connection. Construct a basic_socket_iostream from the supplied socket. Move-construct a basic_socket_iostream from another. 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_after	Set the stream's expiry time relative to now.
expires_at	(Deprecated: Use expiry().) Get the stream's expiry time as an absolute time. Set the stream's expiry time as an absolute time.
expires_from_now	(Deprecated: Use expiry().) Get the stream's expiry time relative to now. (Deprecated: Use expires_after().) Set the stream's expiry time relative to now.

Name	Description
expiry	Get the stream's expiry time as an absolute time.
operator=	Move-assign a basic_socket_iostream from another.
rdbuf	Return a pointer to the underlying streambuf.
socket	Get a reference to the underlying socket.

Requirements

Header: asio/ip/tcp.hpp

Convenience header: asio.hpp

5.179.5 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_context);
...
asio::ip::tcp::no_delay option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.179.6 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

5.179.7 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

5.179.8 ip::tcp::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.179.9 ip::tcp::resolver

The TCP resolver type.

```
typedef basic_resolver< tcp > resolver;
```

Types

Name	Description
endpoint_type	The endpoint type.
executor_type	The type of the executor associated with the object.
flags	A bitmask type (C++ Std [lib.bitmask.types]).
iterator	(Deprecated.) The iterator type.
protocol_type	The protocol type.
query	(Deprecated.) The query type.
results_type	The results type.

Member Functions

Name	Description
async_resolve	(Deprecated.) Asynchronously perform forward resolution of a query to a list of entries. 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. Move-construct a basic_resolver from another.
cancel	Cancel any asynchronous operations that are waiting on the resolver.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context associated with the object.
operator=	Move-assign a basic_resolver from another.
resolve	(Deprecated.) Perform forward resolution of a query to a list of entries. Perform forward resolution of a query to a list of entries. Perform reverse resolution of an endpoint to a list of entries.
~basic_resolver	Destroys the resolver.

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` 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`

5.179.10 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.
executor_type	The type of the executor associated with the 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 <code>basic_socket</code> is always the lowest layer.

Name	Description
message_flags	Bitmask type for flags that can be passed to send and receive operations.
native_handle_type	The native representation of a socket.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</code> associated with the object.
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</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_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.

Name	Description
receive	Receive some data on the socket. Receive some data on a connected socket.
release	Release ownership of the underlying native 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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
write_some	Write some data to the socket.
<code>~basic_stream_socket</code>	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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`

5.179.11 ip::tcp::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.179.12 ip::tcp::v4

Construct to represent the IPv4 TCP protocol.

```
static tcp v4();
```

5.179.13 ip::tcp::v6

Construct to represent the IPv6 TCP protocol.

```
static tcp v6();
```

5.180 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`

5.180.1 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.

Name	Description
basic_endpoint	<p>Default constructor.</p> <p>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.</p> <p>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.</p> <p>Copy constructor.</p> <p>Move constructor.</p>
capacity	Get the capacity of the endpoint in the native type.
data	Get the underlying endpoint in the native type.
operator=	<p>Assign from another endpoint.</p> <p>Move-assign from another endpoint.</p>
port	<p>Get the port associated with the endpoint. The port number is always in the host's byte order.</p> <p>Set the port associated with the endpoint. The port number is always in the host's byte order.</p>
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`

5.180.2 ip::udp::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.180.3 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`

5.180.4 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`

5.180.5 ip::udp::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.180.6 ip::udp::resolver

The UDP resolver type.

```
typedef basic_resolver< udp > resolver;
```

Types

Name	Description
endpoint_type	The endpoint type.
executor_type	The type of the executor associated with the object.
flags	A bitmask type (C++ Std [lib.bitmask.types]).
iterator	(Deprecated.) The iterator type.
protocol_type	The protocol type.
query	(Deprecated.) The query type.
results_type	The results type.

Member Functions

Name	Description
async_resolve	(Deprecated.) Asynchronously perform forward resolution of a query to a list of entries. 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. Move-construct a basic_resolver from another.
cancel	Cancel any asynchronous operations that are waiting on the resolver.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.

Name	Description
get_io_service	(Deprecated: Use get_executor().) Get the io_context associated with the object.
operator=	Move-assign a basic_resolver from another.
resolve	(Deprecated.) Perform forward resolution of a query to a list of entries. Perform forward resolution of a query to a list of entries. Perform reverse resolution of an endpoint to a list of entries.
~basic_resolver	Destroys the resolver.

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.
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` 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`

5.180.7 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.
endpoint_type	The endpoint type.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

Member Functions

Name	Description
<code>assign</code>	Assign an existing native socket to the socket.
<code>async_connect</code>	Start an asynchronous connect.
<code>async_receive</code>	Start an asynchronous receive on a connected socket.
<code>async_receive_from</code>	Start an asynchronous receive.
<code>async_send</code>	Start an asynchronous send on a connected socket.
<code>async_send_to</code>	Start an asynchronous send.
<code>async_wait</code>	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
<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.
<code>basic_datagram_socket</code>	Construct a <code>basic_datagram_socket</code> without opening it. Construct and open a <code>basic_datagram_socket</code> . Construct a <code>basic_datagram_socket</code> , opening it and binding it to the given local endpoint. Construct a <code>basic_datagram_socket</code> on an existing native socket. Move-construct a <code>basic_datagram_socket</code> from another. Move-construct a <code>basic_datagram_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_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</code> associated with the object.
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code>) Get the <code>io_context</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.

Name	Description
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_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.
release	Release ownership of the underlying native socket.
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
~basic_datagram_socket	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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`

5.180.8 ip::udp::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.180.9 ip::udp::v4

Construct to represent the IPv4 UDP protocol.

```
static udp v4();
```

5.180.10 ip::udp::v6

Construct to represent the IPv6 UDP protocol.

```
static udp v6();
```

5.181 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_context);
...
asio::ip::unicast::hops option(4);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
asio::ip::unicast::hops option;
socket.get_option(option);
int ttl = option.value();
```

Requirements

Header: asio/ip/unicast.hpp

Convenience header: asio.hpp

5.182 ip::v4_mapped_t

Tag type used for distinguishing overloads that deal in IPv4-mapped IPv6 addresses.

```
enum v4_mapped_t
```

Values

v4_mapped

Requirements

Header: asio/ip/address_v6.hpp

Convenience header: asio.hpp

5.183 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_context);
...
asio::ip::v6_only option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.184 is_const_buffer_sequence

Trait to determine whether a type satisfies the ConstBufferSequence requirements.

```
template<
    typename T>
struct is_const_buffer_sequence
```

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.185 is_dynamic_buffer

Trait to determine whether a type satisfies the DynamicBuffer requirements.

```
template<
    typename T>
struct is_dynamic_buffer
```

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.186 is_endpoint_sequence

Type trait used to determine whether a type is an endpoint sequence that can be used with with connect and async_connect.

```
template<
    typename T>
struct is_endpoint_sequence
```

Data Members

Name	Description
value	The value member is true if the type may be used as an endpoint sequence.

Requirements

Header: asio/connect.hpp

Convenience header: asio.hpp

5.186.1 is_endpoint_sequence::value

The value member is true if the type may be used as an endpoint sequence.

```
static const bool value;
```

5.187 is_executor

The `is_executor` trait detects whether a type T meets the Executor type requirements.

```
template<
    typename T>
struct is_executor
```

Class template `is_executor` is a UnaryTypeTrait that is derived from `true_type` if the type T meets the syntactic requirements for Executor, otherwise `false_type`.

Requirements

Header: asio/is_executor.hpp

Convenience header: asio.hpp

5.188 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

5.188.1 is_match_condition::value

The value member is true if the type may be used as a match condition.

```
static const bool value;
```

5.189 is Mutable Buffer Sequence

Trait to determine whether a type satisfies the MutableBufferSequence requirements.

```
template<
    typename T>
struct is Mutable Buffer Sequence
```

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.190 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

5.190.1 is Read Buffered::value

The value member is true only if the Stream type supports buffering of read data.

```
static const bool value;
```

5.191 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`

5.191.1 is_write_buffered::value

The value member is true only if the Stream type supports buffering of written data.

```
static const bool value;
```

5.192 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

5.192.1 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);
```

5.192.1.1 local::basic_endpoint::basic_endpoint (1 of 4 overloads)

Default constructor.

```
basic_endpoint();
```

5.192.1.2 local::basic_endpoint::basic_endpoint (2 of 4 overloads)

Construct an endpoint using the specified path name.

```
basic_endpoint(
    const char * path_name);
```

5.192.1.3 local::basic_endpoint::basic_endpoint (3 of 4 overloads)

Construct an endpoint using the specified path name.

```
basic_endpoint(
    const std::string & path_name);
```

5.192.1.4 local::basic_endpoint::basic_endpoint (4 of 4 overloads)

Copy constructor.

```
basic_endpoint(
    const basic_endpoint & other);
```

5.192.2 local::basic_endpoint::capacity

Get the capacity of the endpoint in the native type.

```
std::size_t capacity() const;
```

5.192.3 local::basic_endpoint::data

Get the underlying endpoint in the native type.

```
data_type * data();  
  
const data_type * data() const;
```

5.192.3.1 local::basic_endpoint::data (1 of 2 overloads)

Get the underlying endpoint in the native type.

```
data_type * data();
```

5.192.3.2 local::basic_endpoint::data (2 of 2 overloads)

Get the underlying endpoint in the native type.

```
const data_type * data() const;
```

5.192.4 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

5.192.5 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

5.192.6 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

5.192.7 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.

5.192.8 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

5.192.9 local::basic_endpoint::operator=

Assign from another endpoint.

```
basic_endpoint & operator=(
    const basic_endpoint & other);
```

5.192.10 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

5.192.11 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

5.192.12 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

5.192.13 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);
```

5.192.13.1 local::basic_endpoint::path (1 of 3 overloads)

Get the path associated with the endpoint.

```
std::string path() const;
```

5.192.13.2 local::basic_endpoint::path (2 of 3 overloads)

Set the path associated with the endpoint.

```
void path(  
    const char * p);
```

5.192.13.3 local::basic_endpoint::path (3 of 3 overloads)

Set the path associated with the endpoint.

```
void path(  
    const std::string & p);
```

5.192.14 local::basic_endpoint::protocol

The protocol associated with the endpoint.

```
protocol_type protocol() const;
```

5.192.15 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

5.192.16 local::basic_endpoint::resize

Set the underlying size of the endpoint in the native type.

```
void resize(  
    std::size_t new_size);
```

5.192.17 local::basic_endpoint::size

Get the underlying size of the endpoint in the native type.

```
std::size_t size() const;
```

5.193 local::connect_pair

Create a pair of connected sockets.

```
template<
    typename Protocol>
void connect_pair(
    basic_socket< Protocol > & socket1,
    basic_socket< Protocol > & socket2);

template<
    typename Protocol>
void connect_pair(
    basic_socket< Protocol > & socket1,
    basic_socket< Protocol > & socket2,
   asio::error_code & ec);
```

Requirements

Header: asio/local/connect_pair.hpp

Convenience header: asio.hpp

5.193.1 local::connect_pair (1 of 2 overloads)

Create a pair of connected sockets.

```
template<
    typename Protocol>
void connect_pair(
    basic_socket< Protocol > & socket1,
    basic_socket< Protocol > & socket2);
```

5.193.2 local::connect_pair (2 of 2 overloads)

Create a pair of connected sockets.

```
template<
    typename Protocol>
void connect_pair(
    basic_socket< Protocol > & socket1,
    basic_socket< Protocol > & socket2,
    asio::error_code & ec);
```

5.194 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`

5.194.1 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

5.194.2 local::datagram_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.194.3 local::datagram_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.194.4 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.
endpoint_type	The endpoint type.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.
async_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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_datagram_socket</code>	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.
<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_executor</code>	Get the executor associated with the object.
<code>get_io_context</code>	(Deprecated: Use <code>get_executor()</code>) Get the io_context associated with the object.
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code>) Get the io_context 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_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_datagram_socket from another. Move-assign a basic_datagram_socket from a socket of another protocol type.

Name	Description
receive	Receive some data on a connected socket.
receive_from	Receive a datagram with the endpoint of the sender.
release	Release ownership of the underlying native socket.
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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.
<code>~basic_datagram_socket</code>	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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

5.194.5 local::datagram_protocol::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.195 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

5.195.1 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.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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.
async_wait	Asynchronously wait for the acceptor to become ready to read, ready to write, or to have pending error conditions.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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.

Name	Description
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.
release	Release ownership of the underlying native acceptor.
set_option	Set an option on the acceptor.
wait	Wait for the acceptor to become ready to read, ready to write, or to have pending error conditions.
~basic_socket_acceptor	Destroys the acceptor.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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_context);
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

5.195.2 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`

5.195.3 local::stream_protocol::family

Obtain an identifier for the protocol family.

```
int family() const;
```

5.195.4 local::stream_protocol::iostream

The UNIX domain iostream type.

```
typedef basic_socket_iostream< stream_protocol > iostream;
```

Types

Name	Description
clock_type	The clock type.
duration	The duration type.
duration_type	(Deprecated: Use duration.) The duration type.
endpoint_type	The endpoint type.
protocol_type	The protocol type.
time_point	The time type.
time_type	(Deprecated: Use time_point.) The time type.

Member Functions

Name	Description
basic_socket_iostream	Construct a basic_socket_iostream without establishing a connection. Construct a basic_socket_iostream from the supplied socket. Move-construct a basic_socket_iostream from another. 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_after	Set the stream's expiry time relative to now.
expires_at	(Deprecated: Use expiry().) Get the stream's expiry time as an absolute time. Set the stream's expiry time as an absolute time.
expires_from_now	(Deprecated: Use expiry().) Get the stream's expiry time relative to now. (Deprecated: Use expires_after().) Set the stream's expiry time relative to now.
expiry	Get the stream's expiry time as an absolute time.
operator=	Move-assign a basic_socket_iostream from another.
rdbuf	Return a pointer to the underlying streambuf.
socket	Get a reference to the underlying socket.

Requirements

Header: asio/local/stream_protocol.hpp

Convenience header: asio.hpp

5.195.5 local::stream_protocol::protocol

Obtain an identifier for the protocol.

```
int protocol() const;
```

5.195.6 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.
executor_type	The type of the executor associated with the 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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
protocol_type	The protocol type.
receive_buffer_size	Socket option for the receive buffer size of a socket.

Name	Description
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

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_wait	Asynchronously wait for the socket to become ready to read, ready to write, or to have pending error conditions.
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.

Name	Description
connect	Connect the socket to the specified endpoint.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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_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.
release	Release ownership of the underlying native 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.
wait	Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

Name	Description
write_some	Write some data to the socket.
~basic_stream_socket	Destroys the socket.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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.

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`

5.195.7 local::stream_protocol::type

Obtain an identifier for the type of the protocol.

```
int type() const;
```

5.196 make_work_guard

Create an `executor_work_guard` object.

```

template<
    typename Executor>
executor_work_guard< Executor > make_work_guard(
    const Executor & ex,
    typename enable_if< is_executor< Executor >::value >::type * = 0);

template<
    typename ExecutionContext>
executor_work_guard< typename ExecutionContext::executor_type > make_work_guard(
    ExecutionContext & ctx,
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::type * = 0);

template<
    typename T>
executor_work_guard< typename associated_executor< T >::type > make_work_guard(
    const T & t,
    typename enable_if<!is_executor< T >::value &&!is_convertible< T &, execution_context & >::value >::type * = 0);

template<
    typename T,
    typename Executor>
executor_work_guard< typename associated_executor< T, Executor >::type > make_work_guard(
    const T & t,
    const Executor & ex,
    typename enable_if< is_executor< Executor >::value >::type * = 0);

template<
    typename T,
    typename ExecutionContext>
executor_work_guard< typename associated_executor< T, typename ExecutionContext::executor_type >::type > make_work_guard(
    const T & t,
    ExecutionContext & ctx,
    typename enable_if<!is_executor< T >::value &&!is_convertible< T &, execution_context & >::value >::type * = 0);

```

Requirements

Header: asio/executor_work_guard.hpp

Convenience header: asio.hpp

5.196.1 make_work_guard (1 of 5 overloads)

Create an `executor_work_guard` object.

```

template<
    typename Executor>
executor_work_guard< Executor > make_work_guard(
    const Executor & ex,
    typename enable_if< is_executor< Executor >::value >::type * = 0);

```

5.196.2 make_work_guard (2 of 5 overloads)

Create an `executor_work_guard` object.

```
template<
    typename ExecutionContext>
executor_work_guard< typename ExecutionContext::executor_type > make_work_guard(
    ExecutionContext & ctx,
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::type * = 0);
```

5.196.3 make_work_guard (3 of 5 overloads)

Create an `executor_work_guard` object.

```
template<
    typename T>
executor_work_guard< typename associated_executor< T >::type > make_work_guard(
    const T & t,
    typename enable_if<!is_executor< T >::value &&!is_convertible< T &, execution_context & >::value >::type * = 0);
```

5.196.4 make_work_guard (4 of 5 overloads)

Create an `executor_work_guard` object.

```
template<
    typename T,
    typename Executor>
executor_work_guard< typename associated_executor< T, Executor >::type > make_work_guard(
    const T & t,
    const Executor & ex,
    typename enable_if< is_executor< Executor >::value >::type * = 0);
```

5.196.5 make_work_guard (5 of 5 overloads)

Create an `executor_work_guard` object.

```
template<
    typename T,
    typename ExecutionContext>
executor_work_guard< typename associated_executor< T, typename ExecutionContext::executor_type >::type > make_work_guard(
    const T & t,
    ExecutionContext & ctx,
    typename enable_if<!is_executor< T >::value &&!is_convertible< T &, execution_context & >::value >::type * = 0);
```

5.197 mutable_buffer

Holds a buffer that can be modified.

```
class mutable_buffer
```

Member Functions

Name	Description
data	Get a pointer to the beginning of the memory range.
mutable_buffer	Construct an empty buffer. Construct a buffer to represent a given memory range.
operator+=	Move the start of the buffer by the specified number of bytes.
size	Get the size of the 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 `data()` and `size()` member functions:

```
asio::mutable_buffer b1 = ...;
std::size_t s1 = b1.size();
unsigned char* p1 = static_cast<unsigned char*>(b1.data());
```

The `data()` member 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`

5.197.1 mutable_buffer::data

Get a pointer to the beginning of the memory range.

```
void* data() const;
```

5.197.2 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);
```

5.197.2.1 `mutable_buffer::mutable_buffer (1 of 2 overloads)`

Construct an empty buffer.

```
mutable_buffer();
```

5.197.2.2 `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);
```

5.197.3 `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 n);

mutable_buffer operator+
    std::size_t n,
    const mutable_buffer & b);
```

5.197.3.1 `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 n);
```

5.197.3.2 `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 n,
    const mutable_buffer & b);
```

5.197.4 `mutable_buffer::operator+=`

Move the start of the buffer by the specified number of bytes.

```
mutable_buffer & operator+=(  
    std::size_t n);
```

5.197.5 `mutable_buffer::size`

Get the size of the memory range.

```
std::size_t size() const;
```

5.198 `mutable_buffers_1`

(Deprecated: Use `mutable_buffer`.) 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.
<code>data</code>	Get a pointer to the beginning of the memory range.
<code>end</code>	Get a random-access iterator for one past the last element.
<code>mutable_buffers_1</code>	Construct to represent a given memory range. Construct to represent a single modifiable buffer.
<code>operator+=</code>	Move the start of the buffer by the specified number of bytes.
<code>size</code>	Get the size of the memory range.

Related Functions

Name	Description
<code>operator+</code>	Create a new modifiable buffer that is offset from the start of another.

Requirements

Header: asio/buffer.hpp

Convenience header: asio.hpp

5.198.1 mutable_buffers_1::begin

Get a random-access iterator to the first element.

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

5.198.2 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

5.198.3 mutable_buffers_1::data

Inherited from mutable_buffer.

Get a pointer to the beginning of the memory range.

```
void * data() const;
```

5.198.4 mutable_buffers_1::end

Get a random-access iterator for one past the last element.

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

5.198.5 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);
```

5.198.5.1 `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);
```

5.198.5.2 `mutable_buffers_1::mutable_buffers_1` (2 of 2 overloads)

Construct to represent a single modifiable buffer.

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

5.198.6 `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 n);

mutable_buffer operator+
    std::size_t n,
    const mutable_buffer & b);
```

5.198.6.1 `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 n);
```

5.198.6.2 `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 n,
    const mutable_buffer & b);
```

5.198.7 `mutable_buffers_1::operator+=`

Inherited from mutable_buffer.

Move the start of the buffer by the specified number of bytes.

```
mutable_buffer & operator+=(  
    std::size_t n);
```

5.198.8 mutable_buffers_1::size

Inherited from `mutable_buffer`.

Get the size of the memory range.

```
std::size_t size() const;
```

5.198.9 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>data</code>	Get a pointer to the beginning of the memory range.
<code>mutable_buffer</code>	Construct an empty buffer. Construct a buffer to represent a given memory range.
<code>operator+=</code>	Move the start of the buffer by the specified number of bytes.
<code>size</code>	Get the size of the 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 `data()` and `size()` member functions:

```
asio::mutable_buffer b1 = ...;
std::size_t s1 = b1.size();
unsigned char* p1 = static_cast<unsigned char*>(b1.data());
```

The `data()` member 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`

5.199 null_buffers

(Deprecated: Use the socket/descriptor wait() and async_wait() member functions.) 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

5.199.1 null_buffers::begin

Get a random-access iterator to the first element.

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

5.199.2 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

5.199.3 `null_buffers::end`

Get a random-access iterator for one past the last element.

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

5.199.4 `null_buffers::value_type`

The type for each element in the list of buffers.

```
typedef mutable_buffer value_type;
```

Member Functions

Name	Description
<code>data</code>	Get a pointer to the beginning of the memory range.
<code>mutable_buffer</code>	Construct an empty buffer. Construct a buffer to represent a given memory range.
<code>operator+=</code>	Move the start of the buffer by the specified number of bytes.
<code>size</code>	Get the size of the 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 `data()` and `size()` member functions:

```
asio::mutable_buffer b1 = ...;
std::size_t s1 = b1.size();
unsigned char* p1 = static_cast<unsigned char*>(b1.data());
```

The `data()` member 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`

5.200 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

5.201 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

5.202 placeholders::endpoint

An argument placeholder, for use with boost::bind(), that corresponds to the results argument of a handler for asynchronous functions such as `asio::async_connect`.

```
unspecified endpoint;
```

Requirements

Header: asio/placeholders.hpp

Convenience header: asio.hpp

5.203 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

5.204 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::async_connect`.

```
unspecified iterator;
```

Requirements

Header: `asio/placeholders.hpp`

Convenience header: `asio.hpp`

5.205 placeholders::results

An argument placeholder, for use with `boost::bind()`, that corresponds to the results argument of a handler for asynchronous functions such as `asio::basic_resolver::async_resolve`.

```
unspecified results;
```

Requirements

Header: `asio/placeholders.hpp`

Convenience header: `asio.hpp`

5.206 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`

5.207 posix::descriptor

Provides POSIX descriptor functionality.

```
class descriptor :  
    public posix::descriptor_base
```

Types

Name	Description
bytes_readable	IO control command to get the amount of data that can be read without blocking.
executor_type	The type of the executor associated with the object.
lowest_layer_type	A descriptor is always the lowest layer.
native_handle_type	The native representation of a descriptor.
wait_type	Wait types.

Member Functions

Name	Description
assign	Assign an existing native descriptor to the descriptor.
async_wait	Asynchronously wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.
cancel	Cancel all asynchronous operations associated with the descriptor.
close	Close the descriptor.
descriptor	Construct a descriptor without opening it. Construct a descriptor on an existing native descriptor. Move-construct a descriptor from another.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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 descriptor from another.
release	Release ownership of the native descriptor implementation.
wait	Wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

Protected Member Functions

Name	Description
~descriptor	Protected destructor to prevent deletion through this type.

The `posix::descriptor` class template provides the ability to wrap a POSIX descriptor.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/posix/descriptor.hpp`

Convenience header: `asio.hpp`

5.207.1 posix::descriptor::assign

Assign an existing native descriptor to the descriptor.

```
void assign(
    const native_handle_type & native_descriptor);

void assign(
    const native_handle_type & native_descriptor,
    asio::error_code & ec);
```

5.207.1.1 posix::descriptor::assign (1 of 2 overloads)

Assign an existing native descriptor to the descriptor.

```
void assign(
    const native_handle_type & native_descriptor);
```

5.207.1.2 posix::descriptor::assign (2 of 2 overloads)

Assign an existing native descriptor to the descriptor.

```
void assign(
    const native_handle_type & native_descriptor,
    asio::error_code & ec);
```

5.207.2 posix::descriptor::async_wait

Asynchronously wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

```
template<
    typename WaitHandler>
DEDUCED async_wait(
    wait_type w,
    WaitHandler && handler);
```

This function is used to perform an asynchronous wait for a descriptor to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired descriptor state.

handler The handler to be called when the wait 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_context::post()`.

Example

```
void wait_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Wait succeeded.
    }
}

...

asio::posix::stream_descriptor descriptor(io_context);
...
descriptor.async_wait(
    asio::posix::stream_descriptor::wait_read,
    wait_handler);
```

5.207.3 posix::descriptor::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_context);
...
asio::descriptor_base::bytes_readable command(true);
descriptor.io_control(command);
std::size_t bytes_readable = command.get();
```

Requirements

Header: asio posix descriptor.hpp

Convenience header: asio.hpp

5.207.4 posix::descriptor::cancel

Cancel all asynchronous operations associated with the descriptor.

```
void cancel();

void cancel(
    asio::error_code & ec);
```

5.207.4.1 posix::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.

5.207.4.2 posix::descriptor::cancel (2 of 2 overloads)

Cancel all asynchronous operations associated with the descriptor.

```
void 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.

5.207.5 posix::descriptor::close

Close the descriptor.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.207.5.1 posix::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.

5.207.5.2 posix::descriptor::close (2 of 2 overloads)

Close the descriptor.

```
void 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.

5.207.6 posix::descriptor::descriptor

Construct a descriptor without opening it.

```
explicit descriptor(  
    asio::io_context & io_context);
```

Construct a descriptor on an existing native descriptor.

```
descriptor(  
    asio::io_context & io_context,  
    const native_handle_type & native_descriptor);
```

Move-construct a descriptor from another.

```
descriptor(  
    descriptor && other);
```

5.207.6.1 posix::descriptor::descriptor (1 of 3 overloads)

Construct a descriptor without opening it.

```
descriptor(  
    asio::io_context & io_context);
```

This constructor creates a descriptor without opening it.

Parameters

io_context The `io_context` object that the descriptor will use to dispatch handlers for any asynchronous operations performed on the descriptor.

5.207.6.2 posix::descriptor::descriptor (2 of 3 overloads)

Construct a descriptor on an existing native descriptor.

```
descriptor(  
    asio::io_context & io_context,  
    const native_handle_type & native_descriptor);
```

This constructor creates a descriptor object to hold an existing native descriptor.

Parameters

io_context The `io_context` 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.

5.207.6.3 posix::descriptor::descriptor (3 of 3 overloads)

Move-construct a descriptor from another.

```
descriptor(  
    descriptor && other);
```

This constructor moves a descriptor from one object to another.

Parameters

other The other 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 `descriptor(io_context&)` constructor.

5.207.7 posix::descriptor::executor_type

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio posix descriptor.hpp

Convenience header: asio.hpp

5.207.8 posix::descriptor::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.207.9 posix::descriptor::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.207.10 posix::descriptor::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.207.11 posix::descriptor::io_control

Perform an IO control command on the descriptor.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.207.11.1 posix::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_context);
...
asio::posix::stream_descriptor::bytes_readable command;
descriptor.io_control(command);
std::size_t bytes_readable = command.get();
```

5.207.11.2 posix::descriptor::io_control (2 of 2 overloads)

Perform an IO control command on the descriptor.

```
template<
    typename IoControlCommand>
void 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_context);
...
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();
```

5.207.12 posix::descriptor::is_open

Determine whether the descriptor is open.

```
bool is_open() const;
```

5.207.13 `posix::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;
```

5.207.13.1 `posix::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 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.

5.207.13.2 `posix::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 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.

5.207.14 `posix::descriptor::lowest_layer_type`

A descriptor is always the lowest layer.

```
typedef descriptor lowest_layer_type;
```

Types

Name	Description
bytes_readable	IO control command to get the amount of data that can be read without blocking.
executor_type	The type of the executor associated with the object.
lowest_layer_type	A descriptor is always the lowest layer.
native_handle_type	The native representation of a descriptor. 1146
wait_type	Wait types.

Member Functions

Name	Description
assign	Assign an existing native descriptor to the descriptor.
async_wait	Asynchronously wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.
cancel	Cancel all asynchronous operations associated with the descriptor.
close	Close the descriptor.
descriptor	Construct a descriptor without opening it. Construct a descriptor on an existing native descriptor. Move-construct a descriptor from another.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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 descriptor from another.
release	Release ownership of the native descriptor implementation.
wait	Wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

Protected Member Functions

Name	Description
<code>~descriptor</code>	Protected destructor to prevent deletion through this type.

The `posix::descriptor` class template provides the ability to wrap a POSIX descriptor.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/posix/descriptor.hpp`

Convenience header: `asio.hpp`

5.207.15 posix::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.

5.207.16 posix::descriptor::native_handle_type

The native representation of a descriptor.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: `asio/posix/descriptor.hpp`

Convenience header: `asio.hpp`

5.207.17 posix::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);
```

```
void native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.207.17.1 `posix::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.

5.207.17.2 `posix::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.

5.207.17.3 `posix::descriptor::native_non_blocking` (3 of 3 overloads)

Sets the non-blocking mode of the native descriptor implementation.

```
void 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.

5.207.18 posix::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);

void non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.207.18.1 posix::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`.

5.207.18.2 posix::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`.

5.207.18.3 `posix::descriptor::non_blocking` (3 of 3 overloads)

Sets the non-blocking mode of the descriptor.

```
void 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`.

5.207.19 `posix::descriptor::operator=`

Move-assign a descriptor from another.

```
descriptor & operator=(
    descriptor && other);
```

This assignment operator moves a descriptor from one object to another.

Parameters

other The other 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 `descriptor(io_context&)` constructor.

5.207.20 `posix::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.

5.207.21 posix::descriptor::wait

Wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);  
  
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

5.207.21.1 posix::descriptor::wait (1 of 2 overloads)

Wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);
```

This function is used to perform a blocking wait for a descriptor to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired descriptor state.

Example

Waiting for a descriptor to become readable.

```
asio::posix::stream_descriptor descriptor(io_context);  
...  
descriptor.wait(asio::posix::stream_descriptor::wait_read);
```

5.207.21.2 posix::descriptor::wait (2 of 2 overloads)

Wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

This function is used to perform a blocking wait for a descriptor to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired descriptor state.

ec Set to indicate what error occurred, if any.

Example

Waiting for a descriptor to become readable.

```
asio::posix::stream_descriptor descriptor(io_context);  
...  
asio::error_code ec;  
descriptor.wait(asio::posix::stream_descriptor::wait_read, ec);
```

5.207.22 posix::descriptor::wait_type

Wait types.

```
enum wait_type
```

Values

wait_read Wait for a descriptor to become ready to read.

wait_write Wait for a descriptor to become ready to write.

wait_error Wait for a descriptor to have error conditions pending.

For use with `descriptor::wait()` and `descriptor::async_wait()`.

5.207.23 posix::descriptor::~descriptor

Protected destructor to prevent deletion through this type.

```
~descriptor();
```

This function destroys the descriptor, cancelling any outstanding asynchronous wait operations associated with the descriptor as if by calling `cancel`.

5.208 posix::descriptor_base

The `posix::descriptor_base` class is used as a base for the descriptor class as a 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>wait_type</code>	Wait types.

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

5.208.1 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_context);
...
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

5.208.2 posix::descriptor_base::wait_type

Wait types.

```
enum wait_type
```

Values

wait_read Wait for a descriptor to become ready to read.

wait_write Wait for a descriptor to become ready to write.

wait_error Wait for a descriptor to have error conditions pending.

For use with `descriptor::wait()` and `descriptor::async_wait()`.

5.208.3 posix::descriptor_base::~descriptor_base

Protected destructor to prevent deletion through this type.

```
~descriptor_base();
```

5.209 posix::stream_descriptor

Provides stream-oriented descriptor functionality.

```
class stream_descriptor :
    public posix::descriptor
```

Types

Name	Description
bytes_readable	IO control command to get the amount of data that can be read without blocking.
executor_type	The type of the executor associated with the object.
lowest_layer_type	A descriptor is always the lowest layer.
native_handle_type	The native representation of a descriptor.
wait_type	Wait types.

Member Functions

Name	Description
assign	Assign an existing native descriptor to the descriptor.
async_read_some	Start an asynchronous read.
async_wait	Asynchronously wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.
async_write_some	Start an asynchronous write.
cancel	Cancel all asynchronous operations associated with the descriptor.
close	Close the descriptor.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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 stream_descriptor from another.
read_some	Read some data from the descriptor.
release	Release ownership of the native descriptor implementation.
stream_descriptor	Construct a stream_descriptor without opening it. Construct a stream_descriptor on an existing native descriptor. Move-construct a stream_descriptor from another.
wait	Wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.
write_some	Write some data to the descriptor.

The `posix::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`

5.209.1 posix::stream_descriptor::assign

Assign an existing native descriptor to the descriptor.

```
void assign(
    const native_handle_type & native_descriptor);

void assign(
    const native_handle_type & native_descriptor,
    asio::error_code & ec);
```

5.209.1.1 posix::stream_descriptor::assign (1 of 2 overloads)

Inherited from `posix::descriptor`.

Assign an existing native descriptor to the descriptor.

```
void assign(
    const native_handle_type & native_descriptor);
```

5.209.1.2 posix::stream_descriptor::assign (2 of 2 overloads)

Inherited from `posix::descriptor`.

Assign an existing native descriptor to the descriptor.

```
void assign(
    const native_handle_type & native_descriptor,
    asio::error_code & ec);
```

5.209.2 posix::stream_descriptor::async_read_some

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
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_context::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`.

5.209.3 posix::stream_descriptor::async_wait

Inherited from `posix::descriptor`.

Asynchronously wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

```
template<
    typename WaitHandler>
DEDUCED async_wait(
    wait_type w,
    WaitHandler && handler);
```

This function is used to perform an asynchronous wait for a descriptor to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired descriptor state.

handler The handler to be called when the wait 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_context::post()`.

Example

```
void wait_handler(const asio::error_code& error)
{
    if (!error)
    {
        // Wait succeeded.
    }
}

...

asio::posix::stream_descriptor descriptor(io_context);
...
descriptor.async_wait(
    asio::posix::stream_descriptor::wait_read,
    wait_handler);
```

5.209.4 posix::stream_descriptor::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
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_context::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`.

5.209.5 posix::stream_descriptor::bytes_readable

Inherited from posix::descriptor.

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_context);  
...  
asio::descriptor_base::bytes_readable command(true);  
descriptor.io_control(command);  
std::size_t bytes_readable = command.get();
```

Requirements

Header: `asio/posix/stream_descriptor.hpp`

Convenience header: `asio.hpp`

5.209.6 `posix::stream_descriptor::cancel`

Cancel all asynchronous operations associated with the descriptor.

```
void cancel();  
  
void cancel(  
    asio::error_code & ec);
```

5.209.6.1 `posix::stream_descriptor::cancel (1 of 2 overloads)`

Inherited from posix::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.

5.209.6.2 `posix::stream_descriptor::cancel (2 of 2 overloads)`

Inherited from posix::descriptor.

Cancel all asynchronous operations associated with the descriptor.

```
void 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.

5.209.7 `posix::stream_descriptor::close`

Close the descriptor.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.209.7.1 posix::stream_descriptor::close (1 of 2 overloads)

Inherited from posix::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.

5.209.7.2 posix::stream_descriptor::close (2 of 2 overloads)

Inherited from posix::descriptor.

Close the descriptor.

```
void 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.

5.209.8 posix::stream_descriptor::executor_type

Inherited from posix::descriptor.

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
<code>context</code>	Obtain the underlying execution context.
<code>defer</code>	Request the <code>io_context</code> to invoke the given function object.
<code>dispatch</code>	Request the <code>io_context</code> to invoke the given function object.
<code>on_work_finished</code>	Inform the <code>io_context</code> that some work is no longer outstanding.
<code>on_work_started</code>	Inform the <code>io_context</code> that it has some outstanding work to do.
<code>post</code>	Request the <code>io_context</code> to invoke the given function object.
<code>running_in_this_thread</code>	Determine whether the <code>io_context</code> is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio posix stream_descriptor.hpp

Convenience header: asio.hpp

5.209.9 posix::stream_descriptor::get_executor

Inherited from posix::descriptor.

Get the executor associated with the object.

```
executor_type get_executor();
```

5.209.10 posix::stream_descriptor::get_io_context

Inherited from posix::descriptor.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.209.11 posix::stream_descriptor::get_io_service

Inherited from posix::descriptor.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.209.12 posix::stream_descriptor::io_control

Perform an IO control command on the descriptor.

```
template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command);

template<
    typename IoControlCommand>
void io_control(
    IoControlCommand & command,
    asio::error_code & ec);
```

5.209.12.1 posix::stream_descriptor::io_control (1 of 2 overloads)

Inherited from `posix::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_context);
...
asio::posix::stream_descriptor::bytes_readable command;
descriptor.io_control(command);
std::size_t bytes_readable = command.get();
```

5.209.12.2 posix::stream_descriptor::io_control (2 of 2 overloads)

Inherited from `posix::descriptor`.

Perform an IO control command on the descriptor.

```
template<
    typename IoControlCommand>
void 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_context);
...
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();
```

5.209.13 posix::stream_descriptor::is_open

Inherited from posix::descriptor.

Determine whether the descriptor is open.

```
bool is_open() const;
```

5.209.14 posix::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;
```

5.209.14.1 posix::stream_descriptor::lowest_layer (1 of 2 overloads)

Inherited from posix::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 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.

5.209.14.2 `posix::stream_descriptor::lowest_layer` (2 of 2 overloads)

Inherited from posix::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 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.

5.209.15 `posix::stream_descriptor::lowest_layer_type`

Inherited from posix::descriptor.

A descriptor is always the lowest layer.

```
typedef descriptor lowest_layer_type;
```

Types

Name	Description
bytes_readable	IO control command to get the amount of data that can be read without blocking.
executor_type	The type of the executor associated with the object.
lowest_layer_type	A descriptor is always the lowest layer.
native_handle_type	The native representation of a descriptor.
wait_type	Wait types.

Member Functions

Name	Description
assign	Assign an existing native descriptor to the descriptor.
async_wait	Asynchronously wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.
cancel	Cancel all asynchronous operations associated with the descriptor.
close	Close the descriptor.

Name	Description
descriptor	Construct a descriptor without opening it. Construct a descriptor on an existing native descriptor. Move-construct a descriptor from another.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_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 descriptor from another.
release	Release ownership of the native descriptor implementation.
wait	Wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

Protected Member Functions

Name	Description
~descriptor	Protected destructor to prevent deletion through this type.

The `posix::descriptor` class template provides the ability to wrap a POSIX descriptor.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio posix stream_descriptor.hpp

Convenience header: asio.hpp

5.209.16 posix::stream_descriptor::native_handle

Inherited from posix::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.

5.209.17 posix::stream_descriptor::native_handle_type

Inherited from posix::descriptor.

The native representation of a descriptor.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio posix stream_descriptor.hpp

Convenience header: asio.hpp

5.209.18 posix::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);
```

```
void native_non_blocking(
    bool mode,
    asio::error_code & ec);
```

5.209.18.1 posix::stream_descriptor::native_non_blocking (1 of 3 overloads)

Inherited from posix::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.

5.209.18.2 `posix::stream_descriptor::native_non_blocking` (2 of 3 overloads)

Inherited from `posix::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.

5.209.18.3 `posix::stream_descriptor::native_non_blocking` (3 of 3 overloads)

Inherited from `posix::descriptor`.

Sets the non-blocking mode of the native descriptor implementation.

```
void 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.

5.209.19 `posix::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);

void non_blocking(
    bool mode,
   asio::error_code & ec);
```

5.209.19.1 `posix::stream_descriptor::non_blocking (1 of 3 overloads)`

Inherited from `posix::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`.

5.209.19.2 `posix::stream_descriptor::non_blocking (2 of 3 overloads)`

Inherited from `posix::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`.

5.209.19.3 `posix::stream_descriptor::non_blocking` (3 of 3 overloads)

Inherited from `posix::descriptor`.

Sets the non-blocking mode of the descriptor.

```
void 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`.

5.209.20 `posix::stream_descriptor::operator=`

Move-assign a `posix::stream_descriptor` from another.

```
stream_descriptor & operator=(
    stream_descriptor && other);
```

This assignment operator moves a stream descriptor from one object to another.

Parameters

other The other `posix::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 `stream_descriptor(io_context &)` constructor.

5.209.21 posix::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);
```

5.209.21.1 posix::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`.

5.209.21.2 `posix::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.

5.209.22 `posix::stream_descriptor::release`

Inherited from `posix::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.

5.209.23 `posix::stream_descriptor::stream_descriptor`

Construct a `posix::stream_descriptor` without opening it.

```
explicit stream_descriptor(
    asio::io_context & io_context);
```

Construct a `posix::stream_descriptor` on an existing native descriptor.

```
stream_descriptor(
    asio::io_context & io_context,
    const native_handle_type & native_descriptor);
```

Move-construct a `posix::stream_descriptor` from another.

```
stream_descriptor(
    stream_descriptor && other);
```

5.209.23.1 `posix::stream_descriptor::stream_descriptor` (1 of 3 overloads)

Construct a `posix::stream_descriptor` without opening it.

```
stream_descriptor(  
    asio::io_context & io_context);
```

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_context The `io_context` object that the stream descriptor will use to dispatch handlers for any asynchronous operations performed on the descriptor.

5.209.23.2 `posix::stream_descriptor::stream_descriptor` (2 of 3 overloads)

Construct a `posix::stream_descriptor` on an existing native descriptor.

```
stream_descriptor(  
    asio::io_context & io_context,  
    const native_handle_type & native_descriptor);
```

This constructor creates a stream descriptor object to hold an existing native descriptor.

Parameters

io_context The `io_context` 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.

5.209.23.3 `posix::stream_descriptor::stream_descriptor` (3 of 3 overloads)

Move-construct a `posix::stream_descriptor` from another.

```
stream_descriptor(  
    stream_descriptor && other);
```

This constructor moves a stream descriptor from one object to another.

Parameters

other The other `posix::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 `stream_descriptor(io_context &)` constructor.

5.209.24 posix::stream_descriptor::wait

Wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);  
  
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

5.209.24.1 posix::stream_descriptor::wait (1 of 2 overloads)

Inherited from posix::descriptor.

Wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w);
```

This function is used to perform a blocking wait for a descriptor to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired descriptor state.

Example

Waiting for a descriptor to become readable.

```
asio::posix::stream_descriptor descriptor(io_context);  
...  
descriptor.wait(asio::posix::stream_descriptor::wait_read);
```

5.209.24.2 posix::stream_descriptor::wait (2 of 2 overloads)

Inherited from posix::descriptor.

Wait for the descriptor to become ready to read, ready to write, or to have pending error conditions.

```
void wait(  
    wait_type w,  
    asio::error_code & ec);
```

This function is used to perform a blocking wait for a descriptor to enter a ready to read, write or error condition state.

Parameters

w Specifies the desired descriptor state.

ec Set to indicate what error occurred, if any.

Example

Waiting for a descriptor to become readable.

```
asio::posix::stream_descriptor descriptor(io_context);
...
asio::error_code ec;
descriptor.wait(asio::posix::stream_descriptor::wait_read, ec);
```

5.209.25 posix::stream_descriptor::wait_type

Inherited from posix::descriptor.

Wait types.

```
enum wait_type
```

Values

wait_read Wait for a descriptor to become ready to read.

wait_write Wait for a descriptor to become ready to write.

wait_error Wait for a descriptor to have error conditions pending.

For use with `descriptor::wait()` and `descriptor::async_wait()`.

5.209.26 posix::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);
```

5.209.26.1 posix::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`.

5.209.26.2 posix::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.

5.210 post

Submits a completion token or function object for execution.

```
template<
    typename CompletionToken>
DEDUCED post(
    CompletionToken && token);
```



```
template<
    typename Executor,
    typename CompletionToken>
DEDUCED post(
    const Executor & ex,
    CompletionToken && token,
    typename enable_if< is_executor< Executor >::value >::type * = 0);
```



```
template<
    typename ExecutionContext,
    typename CompletionToken>
DEDUCED post(
    ExecutionContext & ctx,
    CompletionToken && token,
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::type * = 0);
```

Requirements

Header: asio/post.hpp

Convenience header: asio.hpp

5.210.1 post (1 of 3 overloads)

Submits a completion token or function object for execution.

```
template<
    typename CompletionToken>
DEDUCED post(
    CompletionToken && token);
```

This function submits an object for execution using the object's associated executor. The function object is queued for execution, and is never called from the current thread prior to returning from `post()`.

This function has the following effects:

- Constructs a function object handler of type `Handler`, initialized with `handler(forward<CompletionToken>(token))`.
- Constructs an object `result` of type `async_result<Handler>`, initializing the object as `result(handler)`.
- Obtains the handler's associated executor object `ex` by performing `get_associated_executor(handler)`.
- Obtains the handler's associated allocator object `alloc` by performing `get_associated_allocator(handler)`.
- Performs `ex.post(std::move(handler), alloc)`.
- Returns `result.get()`.

5.210.2 post (2 of 3 overloads)

Submits a completion token or function object for execution.

```
template<
    typename Executor,
    typename CompletionToken>
DEDUCED post(
    const Executor & ex,
    CompletionToken && token,
    typename enable_if< is_executor< Executor >::value >::type * = 0);
```

This function submits an object for execution using the specified executor. The function object is queued for execution, and is never called from the current thread prior to returning from `post()`.

This function has the following effects:

- Constructs a function object `handler` of type `Handler`, initialized with `handler(forward<CompletionToken>(token))`.
- Constructs an object `result` of type `async_result<Handler>`, initializing the object as `result(handler)`.
- Obtains the handler's associated executor object `ex1` by performing `get_associated_executor(handler)`.
- Creates a work object `w` by performing `make_work(ex1)`.
- Obtains the handler's associated allocator object `alloc` by performing `get_associated_allocator(handler)`.
- Constructs a function object `f` with a function call operator that performs `ex1.dispatch(std::move(handler), alloc)` followed by `w.reset()`.
- Performs `Executor(ex).post(std::move(f), alloc)`.
- Returns `result.get()`.

5.210.3 post (3 of 3 overloads)

Submits a completion token or function object for execution.

```
template<
    typename ExecutionContext,
    typename CompletionToken>
DEDUCED post(
    ExecutionContext & ctx,
    CompletionToken && token,
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::type * = 0);
```

Return Value

```
post(ctx.get_executor(), forward<CompletionToken>(token)).
```

5.211 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,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >>::value >::type * = ↵
        0);

template<
    typename SyncReadStream,
    typename MutableBufferSequence>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers,
    asio::error_code & ec,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >>::value >::type * = ↵
        0);

template<
    typename SyncReadStream,
    typename MutableBufferSequence,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >>::value >::type * = ↵
        0);

template<
    typename SyncReadStream,
    typename MutableBufferSequence,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    const MutableBufferSequence & buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >>::value >::type * = ↵
        0);

template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >>::value >::type * = 0);

template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read(
    SyncReadStream & s,
    DynamicBuffer && buffers,

```

```

asio::error_code & ec,
typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

template<
    typename SyncReadStream,
    typename DynamicBuffer,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

template<
    typename SyncReadStream,
    typename DynamicBuffer,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

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

5.211.1 read (1 of 12 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,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >::value >::type * = ↵
        0);
```

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());
```

5.211.2 read (2 of 12 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,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >::value >::type * = ↵
        0);
```

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);
```

5.211.3 **read (3 of 12 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,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >::value >::type * = ↵
        0);
```

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.

5.211.4 read (4 of 12 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,
    typename enable_if< is_mutable_buffer_sequence< MutableBufferSequence >::value >::type * = nullptr);
```

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.

5.211.5 `read` (5 of 12 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 specified dynamic buffer sequence 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.

buffers The dynamic buffer sequence 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, buffers,
    asio::transfer_all());
```

5.211.6 read (6 of 12 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    asio::error_code & ec,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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.

buffers The dynamic buffer sequence 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, buffers,
    asio::transfer_all(), ec);
```

5.211.7 read (7 of 12 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 specified dynamic buffer sequence 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.

buffers The dynamic buffer sequence 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.

5.211.8 read (8 of 12 overloads)

Attempt to read a certain amount of data from a stream before returning.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer,
    typename CompletionCondition>
std::size_t read(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 specified dynamic buffer sequence 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.

buffers The dynamic buffer sequence 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.

5.211.9 read (9 of 12 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());
```

5.211.10 read (10 of 12 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);
```

5.211.11 `read` (11 of 12 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.

5.211.12 `read` (12 of 12 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.

5.212 `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

5.212.1 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());
```

5.212.2 `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);
```

5.212.3 **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`.

5.212.4 `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.

5.212.5 `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());
```

5.212.6 `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);
```

5.212.7 `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.

5.212.8 `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.

5.213 `read_until`

Read data into a dynamic buffer sequence, or into a streambuf, until it contains a delimiter, matches a regular expression, or a function object indicates a match.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    char delim);

template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    char delim,
    asio::error_code & ec);

template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    string_view delim);

template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    string_view delim,
    asio::error_code & ec);

template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    const boost::regex & expr);

template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    const boost::regex & expr,
    asio::error_code & ec);
```

```

template<
    typename SyncReadStream,
    typename DynamicBuffer,
    typename MatchCondition>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    MatchCondition match_condition,
    typename enable_if< is_match_condition< MatchCondition >::value >::type * = 0);

template<
    typename SyncReadStream,
    typename DynamicBuffer,
    typename MatchCondition>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    MatchCondition match_condition,
    asio::error_code & ec,
    typename enable_if< is_match_condition< MatchCondition >::value >::type * = 0);

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,
    string_view delim);

template<
    typename SyncReadStream,
    typename Allocator>
std::size_t read_until(
    SyncReadStream & s,
    asio::basic_streambuf< Allocator > & b,
    string_view delim,
    asio::error_code & ec);

template<
    typename SyncReadStream,
    typename Allocator>

```

```

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

5.213.1 read_until (1 of 16 overloads)

Read data into a dynamic buffer sequence until it contains a specified delimiter.

```

template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    char delim);

```

This function is used to read data into the specified dynamic buffer sequence until the dynamic buffer sequence's get area contains the specified delimiter. The call will block until one of the following conditions is true:

- The get area of the dynamic buffer sequence 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 dynamic buffer sequence'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.

buffers The dynamic buffer sequence into which the data will be read.

delim The delimiter character.

Return Value

The number of bytes in the dynamic buffer sequence's get area up to and including the delimiter.

Exceptions

`asio::system_error` Thrown on failure.

Remarks

After a successful `read_until` operation, the dynamic buffer sequence may contain additional data beyond the delimiter. An application will typically leave that data in the dynamic buffer sequence for a subsequent `read_until` operation to examine.

Example

To read data into a `std::string` until a newline is encountered:

```
std::string data;
std::string n = asio::read_until(s,
    asio::dynamic_buffer(data), '\n');
std::string line = data.substr(0, n);
data.erase(0, n);
```

After the `read_until` operation completes successfully, the string `data` contains the delimiter:

```
{ 'a', 'b', ..., 'c', '\n', 'd', 'e', ... }
```

The call to `substr` then extracts the data up to and including the delimiter, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\n' }
```

After the call to `erase`, 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.

5.213.2 `read_until` (2 of 16 overloads)

Read data into a dynamic buffer sequence until it contains a specified delimiter.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    char delim,
    asio::error_code & ec);
```

This function is used to read data into the specified dynamic buffer sequence until the dynamic buffer sequence's get area contains the specified delimiter. The call will block until one of the following conditions is true:

- The get area of the dynamic buffer sequence 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 dynamic buffer sequence'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.

buffers The dynamic buffer sequence 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 dynamic buffer sequence's get area up to and including the delimiter. Returns 0 if an error occurred.

Remarks

After a successful `read_until` operation, the dynamic buffer sequence may contain additional data beyond the delimiter. An application will typically leave that data in the dynamic buffer sequence for a subsequent `read_until` operation to examine.

5.213.3 `read_until` (3 of 16 overloads)

Read data into a dynamic buffer sequence until it contains a specified delimiter.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    string_view delim);
```

This function is used to read data into the specified dynamic buffer sequence until the dynamic buffer sequence's get area contains the specified delimiter. The call will block until one of the following conditions is true:

- The get area of the dynamic buffer sequence 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 dynamic buffer sequence'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.

buffers The dynamic buffer sequence into which the data will be read.

delim The delimiter string.

Return Value

The number of bytes in the dynamic buffer sequence's get area up to and including the delimiter.

Remarks

After a successful `read_until` operation, the dynamic buffer sequence may contain additional data beyond the delimiter. An application will typically leave that data in the dynamic buffer sequence for a subsequent `read_until` operation to examine.

Example

To read data into a `std::string` until a CR-LF sequence is encountered:

```
std::string data;
std::string n = asio::read_until(s,
    asio::dynamic_buffer(data), "\r\n");
std::string line = data.substr(0, n);
data.erase(0, n);
```

After the `read_until` operation completes successfully, the string `data` contains the delimiter:

```
{ 'a', 'b', ..., 'c', '\r', '\n', 'd', 'e', ... }
```

The call to `substr` then extracts the data up to and including the delimiter, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r', '\n' }
```

After the call to `erase`, 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.

5.213.4 `read_until` (4 of 16 overloads)

Read data into a dynamic buffer sequence until it contains a specified delimiter.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    string_view delim,
    asio::error_code & ec);
```

This function is used to read data into the specified dynamic buffer sequence until the dynamic buffer sequence's get area contains the specified delimiter. The call will block until one of the following conditions is true:

- The get area of the dynamic buffer sequence 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 dynamic buffer sequence'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.

buffers The dynamic buffer sequence 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 dynamic buffer sequence's get area up to and including the delimiter. Returns 0 if an error occurred.

Remarks

After a successful `read_until` operation, the dynamic buffer sequence may contain additional data beyond the delimiter. An application will typically leave that data in the dynamic buffer sequence for a subsequent `read_until` operation to examine.

5.213.5 `read_until` (5 of 16 overloads)

Read data into a dynamic buffer sequence until some part of the data it contains matches a regular expression.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    const boost::regex & expr);
```

This function is used to read data into the specified dynamic buffer sequence until the dynamic buffer sequence'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 dynamic buffer sequence'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 dynamic buffer sequence'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.

buffers A dynamic buffer sequence into which the data will be read.

expr The regular expression.

Return Value

The number of bytes in the dynamic buffer sequence'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 dynamic buffer sequence may contain additional data beyond that which matched the regular expression. An application will typically leave that data in the dynamic buffer sequence for a subsequent `read_until` operation to examine.

Example

To read data into a `std::string` until a CR-LF sequence is encountered:

```
std::string data;
std::string n = asio::read_until(s,
    asio::dynamic_buffer(data), boost::regex("\r\n"));
std::string line = data.substr(0, n);
data.erase(0, n);
```

After the `read_until` operation completes successfully, the string `data` contains the delimiter:

```
{ 'a', 'b', ..., 'c', '\r', '\n', 'd', 'e', ... }
```

The call to `substr` then extracts the data up to and including the delimiter, so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r', '\n' }
```

After the call to `erase`, 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.

5.213.6 `read_until` (6 of 16 overloads)

Read data into a dynamic buffer sequence until some part of the data it contains matches a regular expression.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    const boost::regex & expr,
    asio::error_code & ec);
```

This function is used to read data into the specified dynamic buffer sequence until the dynamic buffer sequence'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 dynamic buffer sequence'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 dynamic buffer sequence'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.

buffers A dynamic buffer sequence 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 dynamic buffer sequence'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 dynamic buffer sequence may contain additional data beyond that which matched the regular expression. An application will typically leave that data in the dynamic buffer sequence for a subsequent `read_until` operation to examine.

5.213.7 `read_until` (7 of 16 overloads)

Read data into a dynamic buffer sequence until a function object indicates a match.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer,
    typename MatchCondition>
std::size_t read_until(
    SyncReadStream & s,
    DynamicBuffer && buffers,
    MatchCondition match_condition,
    typename enable_if< is_match_condition< MatchCondition >::value >::type * = 0);
```

This function is used to read data into the specified dynamic buffer sequence until a user-defined match condition function object, when applied to the data contained in the dynamic buffer sequence, 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.

buffers A dynamic buffer sequence 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<typename DynamicBuffer::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 dynamic_buffer'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 dynamic buffer sequence may contain additional data beyond that which matched the function object. An application will typically leave that data in the dynamic buffer sequence for a subsequent 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 read data into a dynamic buffer sequence until whitespace is encountered:

```
typedef asio::buffers_iterator<
    asio::const_buffers_1> 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);
}
...
std::string data;
asio::read_until(s, data, match_whitespace);
```

To read data into a `std::string` 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_;
};

namespace asio {
    template <> struct is_match_condition<match_char>
        : public boost::true_type {};
} // namespace asio
...
std::string data;
asio::read_until(s, data, match_char('a'));
```

5.213.8 `read_until` (8 of 16 overloads)

Read data into a dynamic buffer sequence until a function object indicates a match.

```
template<
    typename SyncReadStream,
    typename DynamicBuffer,
    typename MatchCondition>
std::size_t read_until(
    SyncReadStream & s,
```

```

DynamicBuffer && buffers,
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 dynamic buffer sequence until a user-defined match condition function object, when applied to the data contained in the dynamic buffer sequence, 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.

buffers A dynamic buffer sequence 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<DynamicBuffer::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 dynamic buffer sequence'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 dynamic buffer sequence may contain additional data beyond that which matched the function object. An application will typically leave that data in the dynamic buffer sequence for a subsequent `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.

5.213.9 `read_until` (9 of 16 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 newline (which is discarded), so that the string `line` contains:

```
{ 'a', 'b', ..., 'c' }
```

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.

5.213.10 `read_until` (10 of 16 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.

5.213.11 `read_until` (11 of 16 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,
    string_view 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 newline (which is discarded), so that the string `line` contains:

```
{ 'a', 'b', ..., 'c', '\r' }
```

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.

5.213.12 `read_until` (12 of 16 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,
    string_view 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.

5.213.13 `read_until` (13 of 16 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 newline (which is discarded), so that the string `line` contains:

```
{ 'a', 'b', ... , 'c', '\r' }
```

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.

5.213.14 `read_until` (14 of 16 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.

5.213.15 `read_until` (15 of 16 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 `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 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_;
};

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

5.213.16 `read_until` (16 of 16 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 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.

5.214 resolver_errc::try_again

```
const error::netdb_errors try_again = error::host_not_found_try_again;
```

Requirements

Header: asio/error.hpp

Convenience header: asio.hpp

5.215 serial_port

Provides serial port functionality.

```
class serial_port :  
    public serial_port_base
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
lowest_layer_type	A basic_serial_port is always the lowest layer.
native_handle_type	The native representation of a serial port.

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.
cancel	Cancel all asynchronous operations associated with the serial port.
close	Close the serial port.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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.

Name	Description
native_handle	Get the native serial port representation.
open	Open the serial port using the specified device name.
operator=	Move-assign a serial_port from another.
read_some	Read some data from the serial port.
send_break	Send a break sequence to the serial port.
serial_port	Construct a serial_port without opening it. Construct and open a serial_port. Construct a serial_port on an existing native serial port. Move-construct a serial_port from another.
set_option	Set an option on the serial port.
write_some	Write some data to the serial port.
~serial_port	Destroys the serial port.

The `serial_port` class provides a wrapper over serial port functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/serial_port.hpp`

Convenience header: `asio.hpp`

5.215.1 `serial_port::assign`

Assign an existing native serial port to the serial port.

```
void assign(
    const native_handle_type & native_serial_port);

void assign(
    const native_handle_type & native_serial_port,
    asio::error_code & ec);
```

5.215.1.1 `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);
```

5.215.1.2 serial_port::assign (2 of 2 overloads)

Assign an existing native serial port to the serial port.

```
void assign(
    const native_handle_type & native_serial_port,
    asio::error_code & ec);
```

5.215.2 serial_port::async_read_some

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
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_context::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`.

5.215.3 serial_port::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
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_context::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`.

5.215.4 serial_port::cancel

Cancel all asynchronous operations associated with the serial port.

```
void cancel();  
  
void cancel(  
    asio::error_code & ec);
```

5.215.4.1 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.

5.215.4.2 serial_port::cancel (2 of 2 overloads)

Cancel all asynchronous operations associated with the serial port.

```
void 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.

5.215.5 serial_port::close

Close the serial port.

```
void close();
```

```
void close(  
    asio::error_code & ec);
```

5.215.5.1 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.

5.215.5.2 serial_port::close (2 of 2 overloads)

Close the serial port.

```
void 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.

5.215.6 serial_port::executor_type

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
<code>context</code>	Obtain the underlying execution context.
<code>defer</code>	Request the <code>io_context</code> to invoke the given function object.
<code>dispatch</code>	Request the <code>io_context</code> to invoke the given function object.
<code>on_work_finished</code>	Inform the <code>io_context</code> that some work is no longer outstanding.
<code>on_work_started</code>	Inform the <code>io_context</code> that it has some outstanding work to do.
<code>post</code>	Request the <code>io_context</code> to invoke the given function object.
<code>running_in_this_thread</code>	Determine whether the <code>io_context</code> is running in the current thread.

Friends

Name	Description
<code>operator!=</code>	Compare two executors for inequality.
<code>operator==</code>	Compare two executors for equality.

Requirements

Header: asio/serial_port.hpp

Convenience header: asio.hpp

5.215.7 serial_port::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.215.8 serial_port::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.215.9 serial_port::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.215.10 serial_port::get_option

Get an option from the serial port.

```
template<
    typename GettableSerialPortOption>
void get_option(
    GettableSerialPortOption & option);

template<
    typename GettableSerialPortOption>
void get_option(
    GettableSerialPortOption & option,
    asio::error_code & ec);
```

5.215.10.1 `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.

5.215.10.2 `serial_port::get_option` (2 of 2 overloads)

Get an option from the serial port.

```
template<
    typename GettableSerialPortOption>
void 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.

5.215.11 `serial_port::is_open`

Determine whether the serial port is open.

```
bool is_open() const;
```

5.215.12 `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;
```

5.215.12.1 `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 `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.

5.215.12.2 `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 `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.

5.215.13 `serial_port::lowest_layer_type`

A `basic_serial_port` is always the lowest layer.

```
typedef serial_port lowest_layer_type;
```

Types

Name	Description
<code>executor_type</code>	The type of the executor associated with the 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.

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.

Name	Description
cancel	Cancel all asynchronous operations associated with the serial port.
close	Close the serial port.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_handle	Get the native serial port representation.
open	Open the serial port using the specified device name.
operator=	Move-assign a serial_port from another.
read_some	Read some data from the serial port.
send_break	Send a break sequence to the serial port.
serial_port	Construct a serial_port without opening it. Construct and open a serial_port. Construct a serial_port on an existing native serial port. Move-construct a serial_port from another.
set_option	Set an option on the serial port.
write_some	Write some data to the serial port.
~serial_port	Destroys the serial port.

The `serial_port` class provides a wrapper over serial port functionality.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/serial_port.hpp`

Convenience header: `asio.hpp`

5.215.14 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.

5.215.15 serial_port::native_handle_type

The native representation of a serial port.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/serial_port.hpp

Convenience header: asio.hpp

5.215.16 serial_port::open

Open the serial port using the specified device name.

```
void open(
    const std::string & device);

void open(
    const std::string & device,
    asio::error_code & ec);
```

5.215.16.1 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.

5.215.16.2 `serial_port::open` (2 of 2 overloads)

Open the serial port using the specified device name.

```
void 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.

5.215.17 `serial_port::operator=`

Move-assign a `serial_port` from another.

```
serial_port & operator=(
    serial_port && other);
```

This assignment operator moves a serial port from one object to another.

Parameters

other The other `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 `serial_port(io_context&)` constructor.

5.215.18 `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);
```

5.215.18.1 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`.

5.215.18.2 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.

5.215.19 `serial_port::send_break`

Send a break sequence to the serial port.

```
void send_break();  
  
void send_break(  
    asio::error_code & ec);
```

5.215.19.1 `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.

5.215.19.2 `serial_port::send_break (2 of 2 overloads)`

Send a break sequence to the serial port.

```
void 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.

5.215.20 `serial_port::serial_port`

Construct a `serial_port` without opening it.

```
explicit serial_port(
    asio::io_context & io_context);
```

Construct and open a `serial_port`.

```
explicit serial_port(
    asio::io_context & io_context,
    const char * device);

explicit serial_port(
    asio::io_context & io_context,
    const std::string & device);
```

Construct a `serial_port` on an existing native serial port.

```
serial_port(
    asio::io_context & io_context,
    const native_handle_type & native_serial_port);
```

Move-construct a `serial_port` from another.

```
serial_port(
    serial_port && other);
```

5.215.20.1 `serial_port::serial_port (1 of 5 overloads)`

Construct a `serial_port` without opening it.

```
serial_port(
    asio::io_context & io_context);
```

This constructor creates a serial port without opening it.

Parameters

`io_context` The `io_context` object that the serial port will use to dispatch handlers for any asynchronous operations performed on the port.

5.215.20.2 `serial_port::serial_port (2 of 5 overloads)`

Construct and open a `serial_port`.

```
serial_port(
    asio::io_context & io_context,
    const char * device);
```

This constructor creates and opens a serial port for the specified device name.

Parameters

io_context The `io_context` 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.

5.215.20.3 `serial_port::serial_port (3 of 5 overloads)`

Construct and open a `serial_port`.

```
serial_port(
    asio::io_context & io_context,
    const std::string & device);
```

This constructor creates and opens a serial port for the specified device name.

Parameters

io_context The `io_context` 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.

5.215.20.4 `serial_port::serial_port (4 of 5 overloads)`

Construct a `serial_port` on an existing native serial port.

```
serial_port(
    asio::io_context & io_context,
    const native_handle_type & native_serial_port);
```

This constructor creates a serial port object to hold an existing native serial port.

Parameters

io_context The `io_context` 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.

5.215.20.5 `serial_port::serial_port (5 of 5 overloads)`

Move-construct a `serial_port` from another.

```
serial_port(
    serial_port && other);
```

This constructor moves a serial port from one object to another.

Parameters

other The other `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 `serial_port(io_context&)` constructor.

5.215.21 `serial_port::set_option`

Set an option on the serial port.

```
template<
    typename SettableSerialPortOption>
void set_option(
    const SettableSerialPortOption & option);

template<
    typename SettableSerialPortOption>
void set_option(
    const SettableSerialPortOption & option,
    asio::error_code & ec);
```

5.215.21.1 `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.

5.215.21.2 `serial_port::set_option (2 of 2 overloads)`

Set an option on the serial port.

```
template<
    typename SettableSerialPortOption>
void 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.

5.215.22 `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);
```

5.215.22.1 `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`.

5.215.22.2 `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.

5.215.23 `serial_port::~serial_port`

Destroys the serial port.

```
~serial_port();
```

This function destroys the serial port, cancelling any outstanding asynchronous wait operations associated with the serial port as if by calling `cancel`.

5.216 `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.
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
<code>~serial_port_base</code>	Protected destructor to prevent deletion through this type.

Requirements

Header: asio/serial_port_base.hpp

Convenience header: asio.hpp

5.216.1 serial_port_base::~serial_port_base

Protected destructor to prevent deletion through this type.

```
~serial_port_base();
```

5.217 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

5.217.1 serial_port_base::baud_rate::baud_rate

```
baud_rate(  
    unsigned int rate = 0);
```

5.217.2 serial_port_base::baud_rate::load

```
void load(  
    const ASIO_OPTION_STORAGE & storage,  
    asio::error_code & ec);
```

5.217.3 serial_port_base::baud_rate::store

```
void store(  
    ASIO_OPTION_STORAGE & storage,  
    asio::error_code & ec) const;
```

5.217.4 serial_port_base::baud_rate::value

```
unsigned int value() const;
```

5.218 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

5.218.1 serial_port_base::character_size::character_size

```
character_size(
    unsigned int t = 8);
```

5.218.2 serial_port_base::character_size::load

```
void load(
    const ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec);
```

5.218.3 serial_port_base::character_size::store

```
void store(
    ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec) const;
```

5.218.4 serial_port_base::character_size::value

```
unsigned int value() const;
```

5.219 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

5.219.1 serial_port_base::flow_control::flow_control

```
flow_control(  
    type t = none);
```

5.219.2 serial_port_base::flow_control::load

```
void load(  
    const ASIO_OPTION_STORAGE & storage,  
    asio::error_code & ec);
```

5.219.3 serial_port_base::flow_control::store

```
void store(  
    ASIO_OPTION_STORAGE & storage,  
    asio::error_code & ec) const;
```

5.219.4 serial_port_base::flow_control::type

```
enum type
```

Values

none

software

hardware

5.219.5 serial_port_base::flow_control::value

```
type value() const;
```

5.220 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

5.220.1 serial_port_base::parity::load

```
void load(
    const ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec);
```

5.220.2 serial_port_base::parity::parity

```
parity(
    type t = none);
```

5.220.3 serial_port_base::parity::store

```
void store(
    ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec) const;
```

5.220.4 serial_port_base::parity::type

```
enum type
```

Values

none

odd

even

5.220.5 serial_port_base::parity::value

```
type value() const;
```

5.221 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

5.221.1 serial_port_base::stop_bits::load

```
void load(
    const ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec);
```

5.221.2 serial_port_base::stop_bits::stop_bits

```
stop_bits(
    type t = one);
```

5.221.3 serial_port_base::stop_bits::store

```
void store(
    ASIO_OPTION_STORAGE & storage,
    asio::error_code & ec) const;
```

5.221.4 serial_port_base::stop_bits::type

```
enum type
```

Values

one

onepointfive

two

5.221.5 serial_port_base::stop_bits::value

```
type value() const;
```

5.222 service_already_exists

Exception thrown when trying to add a duplicate service to an [execution_context](#).

```
class service_already_exists
```

Member Functions

Name	Description
service_already_exists	

Requirements

Header: asio/execution_context.hpp

Convenience header: asio.hpp

5.222.1 service_already_exists::service_already_exists

```
service_already_exists();
```

5.223 signal_set

Provides signal functionality.

```
class signal_set
```

Types

Name	Description
executor_type	The type of the executor associated with the object.

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.
cancel	Cancel all operations associated with the signal set.
clear	Remove all signals from a signal_set.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context associated with the object.
remove	Remove a signal from a signal_set.
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.
~signal_set	Destroys the signal set.

The `signal_set` class provides the ability to perform an asynchronous wait for one or more signals to occur.

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_context, 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`

5.223.1 `signal_set::add`

Add a signal to a `signal_set`.

```
void add(
    int signal_number);

void add(
    int signal_number,
    asio::error_code & ec);
```

5.223.1.1 `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.

5.223.1.2 signal_set::add (2 of 2 overloads)

Add a signal to a [signal_set](#).

```
void 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.

5.223.2 signal_set::async_wait

Start an asynchronous operation to wait for a signal to be delivered.

```
template<
    typename SignalHandler>
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_context::post()`.

5.223.3 signal_set::cancel

Cancel all operations associated with the signal set.

```
void cancel();
```

```
void cancel(
    asio::error_code & ec);
```

5.223.3.1 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.

5.223.3.2 signal_set::cancel (2 of 2 overloads)

Cancel all operations associated with the signal set.

```
void 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.

5.223.4 signal_set::clear

Remove all signals from a `signal_set`.

```
void clear();  
  
void clear(  
    asio::error_code & ec);
```

5.223.4.1 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.

5.223.4.2 signal_set::clear (2 of 2 overloads)

Remove all signals from a `signal_set`.

```
void 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.

5.223.5 signal_set::executor_type

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/signal_set.hpp

Convenience header: asio.hpp

5.223.6 signal_set::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.223.7 signal_set::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.223.8 signal_set::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.223.9 signal_set::remove

Remove a signal from a `signal_set`.

```
void remove(
    int signal_number);

void remove(
    int signal_number,
    asio::error_code & ec);
```

5.223.9.1 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.

5.223.9.2 signal_set::remove (2 of 2 overloads)

Remove a signal from a `signal_set`.

```
void 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.

5.223.10 `signal_set::signal_set`

Construct a signal set without adding any signals.

```
explicit signal_set(
    asio::io_context & io_context);
```

Construct a signal set and add one signal.

```
signal_set(
    asio::io_context & io_context,
    int signal_number_1);
```

Construct a signal set and add two signals.

```
signal_set(
    asio::io_context & io_context,
    int signal_number_1,
    int signal_number_2);
```

Construct a signal set and add three signals.

```
signal_set(
    asio::io_context & io_context,
    int signal_number_1,
    int signal_number_2,
    int signal_number_3);
```

5.223.10.1 `signal_set::signal_set (1 of 4 overloads)`

Construct a signal set without adding any signals.

```
signal_set(
    asio::io_context & io_context);
```

This constructor creates a signal set without registering for any signals.

Parameters

io_context The `io_context` object that the signal set will use to dispatch handlers for any asynchronous operations performed on the set.

5.223.10.2 signal_set::signal_set (2 of 4 overloads)

Construct a signal set and add one signal.

```
signal_set(
    asio::io_context & io_context,
    int signal_number_1);
```

This constructor creates a signal set and registers for one signal.

Parameters

io_context The **io_context** 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_context);
signals.add(signal_number_1);
```

5.223.10.3 signal_set::signal_set (3 of 4 overloads)

Construct a signal set and add two signals.

```
signal_set(
    asio::io_context & io_context,
    int signal_number_1,
    int signal_number_2);
```

This constructor creates a signal set and registers for two signals.

Parameters

io_context The **io_context** 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_context);
signals.add(signal_number_1);
signals.add(signal_number_2);
```

5.223.10.4 signal_set::signal_set (4 of 4 overloads)

Construct a signal set and add three signals.

```
signal_set(
    asio::io_context & io_context,
    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_context The `io_context` 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_context);
signals.add(signal_number_1);
signals.add(signal_number_2);
signals.add(signal_number_3);
```

5.223.11 signal_set::~signal_set

Destroys the signal set.

```
~signal_set();
```

This function destroys the signal set, cancelling any outstanding asynchronous wait operations associated with the signal set as if by calling `cancel`.

5.224 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
```

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.
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.
out_of_band_inline	Socket option for putting received out-of-band data inline.
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.
shutdown_type	Different ways a socket may be shutdown.
wait_type	Wait types.

Protected Member Functions

Name	Description
<code>~socket_base</code>	Protected destructor to prevent deletion through this type.

Data Members

Name	Description
max_connections	(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.
max_listen_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

5.224.1 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_context);
...
asio::socket_base::broadcast option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
asio::socket_base::broadcast option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.224.2 `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_context);
...
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

5.224.3 `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_context);
...
asio::socket_base::debug option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::debug option;
socket.get_option(option);
bool is_set = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.224.4 `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_context);
...
asio::socket_base::do_not_route option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::udp::socket socket(io_context);
...
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

5.224.5 `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_context);
...
asio::socket_base::enable_connection_aborted option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.224.6 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_context);
...
asio::socket_base::keep_alive option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.224.7 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_context);
...
asio::socket_base::linger option(true, 30);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.224.8 socket_base::max_connections

(Deprecated: Use max_listen_connections.) The maximum length of the queue of pending incoming connections.

```
static const int max_connections = implementation_defined;
```

5.224.9 socket_base::max_listen_connections

The maximum length of the queue of pending incoming connections.

```
static const int max_listen_connections = implementation_defined;
```

5.224.10 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;
```

5.224.11 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;
```

5.224.12 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

5.224.13 `socket_base::message_out_of_band`

Process out-of-band data.

```
static const int message_out_of_band = implementation_defined;
```

5.224.14 `socket_base::message_peek`

Peek at incoming data without removing it from the input queue.

```
static const int message_peek = implementation_defined;
```

5.224.15 `socket_base::out_of_band_inline`

Socket option for putting received out-of-band data inline.

```
typedef implementation_defined out_of_band_inline;
```

Implements the SOL_SOCKET/SO_OOBINLINE socket option.

Examples

Setting the option:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option(true);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
asio::socket_base::out_of_band_inline option;
socket.get_option(option);
bool value = option.value();
```

Requirements

Header: asio/socket_base.hpp

Convenience header: asio.hpp

5.224.16 `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_context);
...
asio::socket_base::receive_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.224.17 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_context);
...
asio::socket_base::receive_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.224.18 `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_context);
...
asio::socket_base::reuse_address option(true);
acceptor.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::acceptor acceptor(io_context);
...
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

5.224.19 `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_context);
...
asio::socket_base::send_buffer_size option(8192);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.224.20 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_context);
...
asio::socket_base::send_low_watermark option(1024);
socket.set_option(option);
```

Getting the current option value:

```
asio::ip::tcp::socket socket(io_context);
...
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

5.224.21 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.

5.224.22 socket_base::wait_type

Wait types.

```
enum wait_type
```

Values

wait_read Wait for a socket to become ready to read.

wait_write Wait for a socket to become ready to write.

wait_error Wait for a socket to have error conditions pending.

For use with `basic_socket::wait()` and `basic_socket::async_wait()`.

5.224.23 `socket_base::~socket_base`

Protected destructor to prevent deletion through this type.

```
~socket_base();
```

5.225 `spawn`

Start a new stackful coroutine.

```
template<
    typename Function>
void spawn(
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());

template<
    typename Handler,
    typename Function>
void spawn(
    Handler && handler,
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes(),
    typename enable_if<!is_executor< typename decay< Handler >::type >::value &&!is_convertible<
        < Handler &, execution_context & >::value >::type * = 0>;
```



```
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,
    typename Executor>
void spawn(
    const Executor & ex,
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes(),
    typename enable_if< is_executor< Executor >::value >::type * = 0>;
```



```
template<
```

```

typename Function,
typename Executor>
void spawn(
    const strand< Executor > & ex,
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());

template<
    typename Function>
void spawn(
    const asio::io_context::strand & s,
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());

template<
    typename Function,
    typename ExecutionContext>
void spawn(
    ExecutionContext & ctx,
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes(),
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::type * = 0);

```

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

5.225.1 spawn (1 of 7 overloads)

Start a new stackful coroutine, calling the specified handler when it completes.

```
template<
    typename Function>
void spawn(
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());
```

This function is used to launch a new coroutine.

Parameters

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.

5.225.2 spawn (2 of 7 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(),
    typename enable_if<!is_executor< typename decay< Handler >::type >::value &&!is_convertible<
        < Handler &, execution_context & >::value >::type * = 0);
```

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.

5.225.3 `spawn` (3 of 7 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.

5.225.4 `spawn` (4 of 7 overloads)

Start a new stackful coroutine that executes on a given executor.

```
template<
    typename Function,
    typename Executor>
void spawn(
    const Executor & ex,
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes(),
    typename enable_if< is_executor< Executor >::value >::type * = 0);
```

This function is used to launch a new coroutine.

Parameters

ex Identifies the executor that will run the coroutine. The new coroutine is implicitly given its own strand within this executor.

function The coroutine function. The function must have the signature:

```
void function(yield_context yield);
```

attributes Boost.Coroutine attributes used to customise the coroutine.

5.225.5 `spawn` (5 of 7 overloads)

Start a new stackful coroutine that executes on a given strand.

```
template<
    typename Function,
    typename Executor>
void spawn(
    const strand< Executor > & ex,
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());
```

This function is used to launch a new coroutine.

Parameters

ex Identifies the strand that will run the coroutine.

function The coroutine function. The function must have the signature:

```
void function(yield_context yield);
```

attributes Boost.Coroutine attributes used to customise the coroutine.

5.225.6 `spawn` (6 of 7 overloads)

Start a new stackful coroutine that executes in the context of a strand.

```
template<
    typename Function>
void spawn(
    const asio::io_context::strand & s,
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes());
```

This function is used to launch a new coroutine.

Parameters

s 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.

5.225.7 `spawn` (7 of 7 overloads)

Start a new stackful coroutine that executes on a given execution context.

```

template<
    typename Function,
    typename ExecutionContext>
void spawn(
    ExecutionContext & ctx,
    Function && function,
    const boost::coroutines::attributes & attributes = boost::coroutines::attributes(),
    typename enable_if< is_convertible< ExecutionContext &, execution_context & >::value >::value_type * = 0);

```

This function is used to launch a new coroutine.

Parameters

ctx Identifies the execution context that will run the coroutine. The new coroutine is implicitly given its own strand within this execution context.

function The coroutine function. The function must have the signature:

```
void function(yield_context yield);
```

attributes Boost.Coroutine attributes used to customise the coroutine.

5.226 ssl::context

```

class context :
public ssl::context_base,
noncopyable

```

Types

Name	Description
file_format	File format types.
method	Different methods supported by a context.
native_handle_type	The native handle type of the SSL context.
options	Bitmask type for SSL options.
password_purpose	Purpose of PEM password.

Member Functions

Name	Description
add_certificate_authority	Add certification authority for performing verification.
add_verify_path	Add a directory containing certificate authority files to be used for performing verification.

Name	Description
clear_options	Clear options on the context.
context	Constructor. Move-construct a context from another.
load_verify_file	Load a certification authority file for performing verification.
native_handle	Get the underlying implementation in the native type.
operator=	Move-assign a context from another.
set_default_verify_paths	Configures the context to use the default directories for finding certification authority certificates.
set_options	Set options on the context.
set_password_callback	Set the password callback.
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.
use_certificate	Use a certificate from a memory buffer.
use_certificate_chain	Use a certificate chain from a memory buffer.
use_certificate_chain_file	Use a certificate chain from a file.
use_certificate_file	Use a certificate from a file.
use_private_key	Use a private key from a memory buffer.
use_private_key_file	Use a private key from a file.
use_rsa_private_key	Use an RSA private key from a memory buffer.
use_rsa_private_key_file	Use an RSA private key from a file.
use_tmp_dh	Use the specified memory buffer to obtain the temporary Diffie-Hellman parameters.
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_tls1	Disable TLS v1.
no_tls1_1	Disable TLS v1.1.
no_tls1_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

5.226.1 ssl::context::add_certificate_authority

Add certification authority for performing verification.

```
void add_certificate_authority(
    const const_buffer & ca);

void add_certificate_authority(
    const const_buffer & ca,
    asio::error_code & ec);
```

5.226.1.1 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.

5.226.1.2 ssl::context::add_certificate_authority (2 of 2 overloads)

Add certification authority for performing verification.

```
void 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.

5.226.2 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);

void add_verify_path(
    const std::string & path,
    asio::error_code & ec);
```

5.226.2.1 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.

5.226.2.2 ssl::context::add_verify_path (2 of 2 overloads)

Add a directory containing certificate authority files to be used for performing verification.

```
void 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.

5.226.3 ssl::context::clear_options

Clear options on the context.

```
void clear_options(
    options o);

void clear_options(
    options o,
    asio::error_code & ec);
```

5.226.3.1 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.

5.226.3.2 ssl::context::clear_options (2 of 2 overloads)

Clear options on the context.

```
void 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.

5.226.4 ssl::context::context

Constructor.

```
explicit context(
    method m);
```

Move-construct a context from another.

```
context(
    context && other);
```

5.226.4.1 ssl::context::context (1 of 2 overloads)

Constructor.

```
context(
    method m);
```

5.226.4.2 ssl::context::context (2 of 2 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.

5.226.5 `ssl::context::default_workarounds`

Implement various bug workarounds.

```
static const long default_workarounds = implementation_defined;
```

5.226.6 `ssl::context::file_format`

File format types.

```
enum file_format
```

Values

asn1 ASN.1 file.

pem PEM file.

5.226.7 `ssl::context::load_verify_file`

Load a certification authority file for performing verification.

```
void load_verify_file(  
    const std::string & filename);  
  
void load_verify_file(  
    const std::string & filename,  
    asio::error_code & ec);
```

5.226.7.1 `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.

5.226.7.2 **ssl::context::load_verify_file (2 of 2 overloads)**

Load a certification authority file for performing verification.

```
void 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.

5.226.8 **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.

tls Generic TLS.

tls_client TLS client.

tls_server TLS server.

5.226.9 `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.

5.226.10 `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`

5.226.11 `ssl::context::no_compression`

Disable compression. Compression is disabled by default.

```
static const long no_compression = implementation_defined;
```

5.226.12 `ssl::context::no_sslv2`

Disable SSL v2.

```
static const long no_sslv2 = implementation_defined;
```

5.226.13 `ssl::context::no_sslv3`

Disable SSL v3.

```
static const long no_sslv3 = implementation_defined;
```

5.226.14 `ssl::context::no_tlsv1`

Disable TLS v1.

```
static const long no_tlsv1 = implementation_defined;
```

5.226.15 `ssl::context::no_tlsv1_1`

Disable TLS v1.1.

```
static const long no_tlsv1_1 = implementation_defined;
```

5.226.16 `ssl::context::no_tlsv1_2`

Disable TLS v1.2.

```
static const long no_tlsv1_2 = implementation_defined;
```

5.226.17 `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.

5.226.18 `ssl::context::options`

Bitmask type for SSL options.

```
typedef long options;
```

Requirements

Header: asio/ssl/context.hpp

Convenience header: asio/ssl.hpp

5.226.19 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.

5.226.20 ssl::context::set_default_verify_paths

Configures the context to use the default directories for finding certification authority certificates.

```
void set_default_verify_paths();  
  
void set_default_verify_paths(  
    asio::error_code & ec);
```

5.226.20.1 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.

5.226.20.2 ssl::context::set_default_verify_paths (2 of 2 overloads)

Configures the context to use the default directories for finding certification authority certificates.

```
void 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.

5.226.21 `ssl::context::set_options`

Set options on the context.

```
void set_options(
    options o);

void set_options(
    options o,
    asio::error_code & ec);
```

5.226.21.1 `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.

5.226.21.2 `ssl::context::set_options (2 of 2 overloads)`

Set options on the context.

```
void 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`.

5.226.22 `ssl::context::set_password_callback`

Set the password callback.

```
template<
    typename PasswordCallback>
void set_password_callback(
    PasswordCallback callback);

template<
    typename PasswordCallback>
void set_password_callback(
    PasswordCallback callback,
    asio::error_code & ec);
```

5.226.22.1 `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.

5.226.22.2 ssl::context::set_password_callback (2 of 2 overloads)

Set the password callback.

```
template<
    typename PasswordCallback>
void 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.

5.226.23 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>
void set_verify_callback(
    VerifyCallback callback,
    asio::error_code & ec);
```

5.226.23.1 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.

5.226.23.2 ssl::context::set_verify_callback (2 of 2 overloads)

Set the callback used to verify peer certificates.

```
template<
    typename VerifyCallback>
void 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.

5.226.24 ssl::context::set_verify_depth

Set the peer verification depth.

```
void set_verify_depth(
    int depth);

void set_verify_depth(
    int depth,
    asio::error_code & ec);
```

5.226.24.1 `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`.

5.226.24.2 `ssl::context::set_verify_depth` (2 of 2 overloads)

Set the peer verification depth.

```
void 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`.

5.226.25 `ssl::context::set_verify_mode`

Set the peer verification mode.

```
void set_verify_mode(  
    verify_mode v);
```

```
void set_verify_mode(  
    verify_mode v,  
    asio::error_code & ec);
```

5.226.25.1 `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`.

5.226.25.2 `ssl::context::set_verify_mode` (2 of 2 overloads)

Set the peer verification mode.

```
void 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`.

5.226.26 `ssl::context::single_dh_use`

Always create a new key when using `tmp_dh` parameters.

```
static const long single_dh_use = implementation_defined;
```

5.226.27 `ssl::context::use_certificate`

Use a certificate from a memory buffer.

```
void use_certificate(
    const const_buffer & certificate,
    file_format format);

void use_certificate(
    const const_buffer & certificate,
    file_format format,
    asio::error_code & ec);
```

5.226.27.1 `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.

5.226.27.2 `ssl::context::use_certificate (2 of 2 overloads)`

Use a certificate from a memory buffer.

```
void 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.

5.226.28 `ssl::context::use_certificate_chain`

Use a certificate chain from a memory buffer.

```
void use_certificate_chain(
    const const_buffer & chain);

void use_certificate_chain(
    const const_buffer & chain,
    asio::error_code & ec);
```

5.226.28.1 `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.

5.226.28.2 `ssl::context::use_certificate_chain (2 of 2 overloads)`

Use a certificate chain from a memory buffer.

```
void 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.

5.226.29 `ssl::context::use_certificate_chain_file`

Use a certificate chain from a file.

```
void use_certificate_chain_file(
    const std::string & filename);
```

```
void use_certificate_chain_file(
    const std::string & filename,
    asio::error_code & ec);
```

5.226.29.1 `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.

5.226.29.2 `ssl::context::use_certificate_chain_file (2 of 2 overloads)`

Use a certificate chain from a file.

```
void 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.

5.226.30 ssl::context::use_certificate_file

Use a certificate from a file.

```
void use_certificate_file(
    const std::string & filename,
    file_format format);

void use_certificate_file(
    const std::string & filename,
    file_format format,
    asio::error_code & ec);
```

5.226.30.1 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.

5.226.30.2 ssl::context::use_certificate_file (2 of 2 overloads)

Use a certificate from a file.

```
void 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.

5.226.31 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);

void use_private_key(
    const const_buffer & private_key,
    file_format format,
    asio::error_code & ec);
```

5.226.31.1 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.

5.226.31.2 `ssl::context::use_private_key` (2 of 2 overloads)

Use a private key from a memory buffer.

```
void 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.

5.226.32 `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);

void use_private_key_file(
    const std::string & filename,
    file_format format,
    asio::error_code & ec);
```

5.226.32.1 `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.

5.226.32.2 ssl::context::use_private_key_file (2 of 2 overloads)

Use a private key from a file.

```
void 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.

5.226.33 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);

void use_rsa_private_key(
    const const_buffer & private_key,
    file_format format,
    asio::error_code & ec);
```

5.226.33.1 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.

5.226.33.2 **ssl::context::use_rsa_private_key (2 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,
    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.

5.226.34 **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);

void use_rsa_private_key_file(
    const std::string & filename,
    file_format format,
    asio::error_code & ec);
```

5.226.34.1 **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.

5.226.34.2 ssl::context::use_rsa_private_key_file (2 of 2 overloads)

Use an RSA private key from a file.

```
void 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.

5.226.35 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);

void use_tmp_dh (
    const const_buffer & dh,
    asio::error_code & ec);
```

5.226.35.1 `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`.

5.226.35.2 `ssl::context::use_tmp_dh` (2 of 2 overloads)

Use the specified memory buffer to obtain the temporary Diffie-Hellman parameters.

```
void 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`.

5.226.36 `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);

void use_tmp_dh_file(
    const std::string & filename,
    asio::error_code & ec);
```

5.226.36.1 `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`.

5.226.36.2 `ssl::context::use_tmp_dh_file (2 of 2 overloads)`

Use the specified file to obtain the temporary Diffie-Hellman parameters.

```
void 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`.

5.226.37 `ssl::context::~context`

Destructor.

```
~context();
```

5.227 `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

5.227.1 ssl::context_base::default_workarounds

Implement various bug workarounds.

```
static const long default_workarounds = implementation_defined;
```

5.227.2 `ssl::context_base::file_format`

File format types.

```
enum file_format
```

Values

asn1 ASN.1 file.

pem PEM file.

5.227.3 `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.

tls Generic TLS.

tls_client TLS client.

tls_server TLS server.

5.227.4 `ssl::context_base::no_compression`

Disable compression. Compression is disabled by default.

```
static const long no_compression = implementation_defined;
```

5.227.5 `ssl::context_base::no_sslv2`

Disable SSL v2.

```
static const long no_sslv2 = implementation_defined;
```

5.227.6 `ssl::context_base::no_sslv3`

Disable SSL v3.

```
static const long no_sslv3 = implementation_defined;
```

5.227.7 `ssl::context_base::no_tlsv1`

Disable TLS v1.

```
static const long no_tlsv1 = implementation_defined;
```

5.227.8 `ssl::context_base::no_tlsv1_1`

Disable TLS v1.1.

```
static const long no_tlsv1_1 = implementation_defined;
```

5.227.9 `ssl::context_base::no_tlsv1_2`

Disable TLS v1.2.

```
static const long no_tlsv1_2 = implementation_defined;
```

5.227.10 `ssl::context_base::options`

Bitmask type for SSL options.

```
typedef long options;
```

Requirements

Header: `asio/ssl/context_base.hpp`

Convenience header: `asio/ssl.hpp`

5.227.11 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.

5.227.12 ssl::context_base::single_dh_use

Always create a new key when using tmp_dh parameters.

```
static const long single_dh_use = implementation_defined;
```

5.227.13 ssl::context_base::~context_base

Protected destructor to prevent deletion through this type.

```
~context_base();
```

5.228 ssl::error::get_stream_category

```
const asio::error_category & get_stream_category();
```

Requirements

Header: asio/ssl/error.hpp

Convenience header: asio/ssl.hpp

5.229 ssl::error::make_error_code

```
asio::error_code make_error_code(  
    stream_errors e);
```

Requirements

Header: asio/ssl/error.hpp

Convenience header: asio/ssl.hpp

5.230 ssl::error::stream_category

```
static const asio::error_category & stream_category = asio::ssl::error::get_stream_category();
```

Requirements

Header: asio/ssl/error.hpp

Convenience header: asio/ssl.hpp

5.231 ssl::error::stream_errors

```
enum stream_errors
```

Values

stream_truncated The underlying stream closed before the ssl stream gracefully shut down.

Requirements

Header: asio/ssl/error.hpp

Convenience header: asio/ssl.hpp

5.232 ssl::rfc2818_verification

Verifies a certificate against a hostname according to the rules described in RFC 2818.

```
class rfc2818_verification
```

Types

Name	Description
result_type	The type of the function object's result.

Member Functions

Name	Description
operator()	Perform certificate verification.
rfc2818_verification	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_context io_context;
ssl_socket sock(io_context, ctx);
tcp::resolver resolver(io_context);
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

5.232.1 ssl::rfc2818_verification::operator()

Perform certificate verification.

```

bool operator()
    bool preverified,
    verify_context & ctx) const;

```

5.232.2 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

5.232.3 ssl::rfc2818_verification::rfc2818_verification

Constructor.

```

rfc2818_verification(
    const std::string & host);

```

5.233 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.
executor_type	The type of the executor associated with the object.
handshake_type	Different handshake types.
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_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context associated with the object.
handshake	Perform SSL handshaking.
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.

Name	Description
read_some	Read some data from the stream.
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_context io_context;
asio::ssl::context ctx(asio::ssl::context::sslv23);
asio::ssl::stream<asio::ip::tcp::socket> sock(io_context, ctx);
```

Requirements

Header: `asio/ssl/stream.hpp`

Convenience header: `asio/ssl.hpp`

5.233.1 `ssl::stream::async_handshake`

Start an asynchronous SSL handshake.

```
template<
    typename HandshakeHandler>
DEDUCED async_handshake(
    handshake_type type,
    HandshakeHandler && handler);

template<
    typename ConstBufferSequence,
    typename BufferedHandshakeHandler>
```

```
DEDUCED async_handshake(
    handshake_type type,
    const ConstBufferSequence & buffers,
    BufferedHandshakeHandler && handler);
```

5.233.1.1 `ssl::stream::async_handshake` (1 of 2 overloads)

Start an asynchronous SSL handshake.

```
template<
    typename HandshakeHandler>
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.
);
```

5.233.1.2 `ssl::stream::async_handshake` (2 of 2 overloads)

Start an asynchronous SSL handshake.

```
template<
    typename ConstBufferSequence,
    typename BufferedHandshakeHandler>
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.
);
```

5.233.2 `ssl::stream::async_read_some`

Start an asynchronous read.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
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.

5.233.3 `ssl::stream::async_shutdown`

Asynchronously shut down SSL on the stream.

```
template<
    typename ShutdownHandler>
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.
);
```

5.233.4 `ssl::stream::async_write_some`

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
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.

5.233.5 `ssl::stream::executor_type`

The type of the executor associated with the object.

```
typedef lowest_layer_type::executor_type executor_type;
```

Requirements

Header: `asio/ssl/stream.hpp`

Convenience header: `asio/ssl.hpp`

5.233.6 `ssl::stream::get_executor`

Get the executor associated with the object.

```
executor_type get_executor();
```

This function may be used to obtain the executor object that the stream uses to dispatch handlers for asynchronous operations.

Return Value

A copy of the executor that stream will use to dispatch handlers.

5.233.7 `ssl::stream::get_io_context`

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

5.233.8 `ssl::stream::get_io_service`

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

5.233.9 `ssl::stream::handshake`

Perform SSL handshaking.

```
void handshake(
    handshake_type type);

void handshake(
    handshake_type type,
    asio::error_code & ec);

template<
    typename ConstBufferSequence>
void handshake(
    handshake_type type,
    const ConstBufferSequence & buffers);

template<
    typename ConstBufferSequence>
void handshake(
    handshake_type type,
    const ConstBufferSequence & buffers,
    asio::error_code & ec);
```

5.233.9.1 `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.

5.233.9.2 **ssl::stream::handshake (2 of 4 overloads)**

Perform SSL handshaking.

```
void 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.

5.233.9.3 **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.

5.233.9.4 **ssl::stream::handshake (4 of 4 overloads)**

Perform SSL handshaking.

```
template<
    typename ConstBufferSequence>
void 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.

5.233.10 `ssl::stream::handshake_type`

Different handshake types.

```
enum handshake_type
```

Values

client Perform handshaking as a client.

server Perform handshaking as a server.

5.233.11 `ssl::stream::lowest_layer`

Get a reference to the lowest layer.

```
lowest_layer_type & lowest_layer();  
  
const lowest_layer_type & lowest_layer() const;
```

5.233.11.1 `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.

5.233.11.2 `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.

5.233.12 `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

5.233.13 `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_context, 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)
    {
        // ...
    }
}
```

5.233.14 `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

5.233.15 `ssl::stream::next_layer`

Get a reference to the next layer.

```
const next_layer_type & next_layer() const;
```

```
next_layer_type & next_layer();
```

5.233.15.1 `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.

5.233.15.2 `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.

5.233.16 `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`

5.233.17 `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);
```

5.233.17.1 `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.

5.233.17.2 `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.

5.233.18 `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>
void set_verify_callback(
    VerifyCallback callback,
    asio::error_code & ec);
```

5.233.18.1 `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`.

5.233.18.2 `ssl::stream::set_verify_callback` (2 of 2 overloads)

Set the callback used to verify peer certificates.

```
template<
    typename VerifyCallback>
void 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`.

5.233.19 `ssl::stream::set_verify_depth`

Set the peer verification depth.

```
void set_verify_depth(
    int depth);

void set_verify_depth(
    int depth,
    asio::error_code & ec);
```

5.233.19.1 `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`.

5.233.19.2 `ssl::stream::set_verify_depth (2 of 2 overloads)`

Set the peer verification depth.

```
void 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`.

5.233.20 `ssl::stream::set_verify_mode`

Set the peer verification mode.

```
void set_verify_mode(
    verify_mode v);

void set_verify_mode(
    verify_mode v,
    asio::error_code & ec);
```

5.233.20.1 `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.

5.233.20.2 **ssl::stream::set_verify_mode (2 of 2 overloads)**

Set the peer verification mode.

```
void 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.

5.233.21 **ssl::stream::shutdown**

Shut down SSL on the stream.

```
void shutdown();

void shutdown(
    asio::error_code & ec);
```

5.233.21.1 **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.

5.233.21.2 `ssl::stream::shutdown` (2 of 2 overloads)

Shut down SSL on the stream.

```
void 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.

5.233.22 `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.

5.233.23 `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);
```

5.233.23.1 `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.

5.233.23.2 `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.

5.233.24 `ssl::stream::~stream`

Destructor.

```
~stream();
```

Remarks

A `stream` object must not be destroyed while there are pending asynchronous operations associated with it.

5.234 `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`

5.234.1 `ssl::stream::impl_struct::ssl`

```
SSL * ssl;
```

5.235 `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

5.235.1 ssl::stream_base::handshake_type

Different handshake types.

```
enum handshake_type
```

Values

client Perform handshaking as a client.

server Perform handshaking as a server.

5.235.2 ssl::stream_base::~stream_base

Protected destructor to prevent deletion through this type.

```
~stream_base();
```

5.236 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

5.237 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
native_handle_type	The native handle type of the verification context.

Member Functions

Name	Description
native_handle	Get the underlying implementation in the native type.
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`

5.237.1 `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.

5.237.2 `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`

5.237.3 `ssl::verify_context::verify_context`

Constructor.

```
verify_context(
    native_handle_type handle);
```

5.238 `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

5.239 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

5.240 ssl::verify_none

No verification.

```
const int verify_none = implementation_defined;
```

Requirements

Header: asio/ssl/verify_mode.hpp

Convenience header: asio/ssl.hpp

5.241 ssl::verify_peer

Verify the peer.

```
const int verify_peer = implementation_defined;
```

Requirements

Header: asio/ssl/verify_mode.hpp

Convenience header: asio/ssl.hpp

5.242 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.
executor_type	The type of the executor associated with the object.
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. Move-construct a basic_waitable_timer from another.
cancel	Cancel any asynchronous operations that are waiting on the timer. (Deprecated: Use non-error_code overload.) Cancel any asynchronous operations that are waiting on the timer.
cancel_one	Cancels one asynchronous operation that is waiting on the timer. (Deprecated: Use non-error_code overload.) Cancels one asynchronous operation that is waiting on the timer.
expires_after	Set the timer's expiry time relative to now.
expires_at	(Deprecated: Use expiry().) Get the timer's expiry time as an absolute time. Set the timer's expiry time as an absolute time. (Deprecated: Use non-error_code overload.) Set the timer's expiry time as an absolute time.
expires_from_now	(Deprecated: Use expiry().) Get the timer's expiry time relative to now. (Deprecated: Use expires_after().) Set the timer's expiry time relative to now.
expiry	Get the timer's expiry time as an absolute time.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.

Name	Description
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code> .) Get the <code>io_context</code> associated with the object.
<code>operator=</code>	Move-assign a <code>basic_waitable_timer</code> from another.
<code>wait</code>	Perform a blocking wait on the timer.
<code>~basic_waitable_timer</code>	Destroys the timer.

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_context);

// Set an expiry time relative to now.
timer.expires_after(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_context,
    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_after(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_after()` 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: `asio.hpp`

5.243 strand

Provides serialised function invocation for any executor type.

```
template<
    typename Executor>
class strand
```

Types

Name	Description
inner_executor_type	The type of the underlying executor.

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the strand to invoke the given function object.
dispatch	Request the strand to invoke the given function object.
get_inner_executor	Obtain the underlying executor.
on_work_finished	Inform the strand that some work is no longer outstanding.
on_work_started	Inform the strand that it has some outstanding work to do.
operator=	Assignment operator. Converting assignment operator. Move assignment operator. Converting move assignment operator.
post	Request the strand to invoke the given function object.
running_in_this_thread	Determine whether the strand is running in the current thread.
strand	Default constructor. Construct a strand for the specified executor. Copy constructor. Converting constructor. Move constructor. Converting move constructor.
~strand	Destructor.

Friends

Name	Description
operator!=	Compare two strands for inequality.
operator==	Compare two strands for equality.

Requirements

Header: asio/strand.hpp

Convenience header: asio.hpp

5.243.1 strand::context

Obtain the underlying execution context.

```
execution_context & context() const;
```

5.243.2 strand::defer

Request the strand to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void defer(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the executor to execute the given function object. The function object will never be executed inside this function. Instead, it will be scheduled by the underlying executor's `defer()` function.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.243.3 strand::dispatch

Request the strand to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void dispatch(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the strand to execute the given function object on its underlying executor. The function object will be executed inside this function if the strand is not otherwise busy and if the underlying executor's `dispatch()` function is also able to execute the function before returning.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.243.4 strand::get_inner_executor

Obtain the underlying executor.

```
inner_executor_type get_inner_executor() const;
```

5.243.5 strand::inner_executor_type

The type of the underlying executor.

```
typedef Executor inner_executor_type;
```

Requirements

Header: asio/strand.hpp

Convenience header: asio.hpp

5.243.6 strand::on_work_finished

Inform the strand that some work is no longer outstanding.

```
void on_work_finished() const;
```

The strand delegates this call to its underlying executor.

5.243.7 strand::on_work_started

Inform the strand that it has some outstanding work to do.

```
void on_work_started() const;
```

The strand delegates this call to its underlying executor.

5.243.8 strand::operator!=

Compare two strands for inequality.

```
friend bool operator!=(
    const strand & a,
    const strand & b);
```

Two strands are equal if they refer to the same ordered, non-concurrent state.

Requirements

Header: asio/strand.hpp

Convenience header: asio.hpp

5.243.9 strand::operator=

Assignment operator.

```
strand & operator=(  
    const strand & other);
```

Converting assignment operator.

```
template<  
    class OtherExecutor>  
strand & operator=(  
    const strand< OtherExecutor > & other);
```

Move assignment operator.

```
strand & operator=(  
    strand && other);
```

Converting move assignment operator.

```
template<  
    class OtherExecutor>  
strand & operator=(  
    const strand< OtherExecutor > && other);
```

5.243.9.1 strand::operator= (1 of 4 overloads)

Assignment operator.

```
strand & operator=(  
    const strand & other);
```

5.243.9.2 strand::operator= (2 of 4 overloads)

Converting assignment operator.

```
template<  
    class OtherExecutor>  
strand & operator=(  
    const strand< OtherExecutor > & other);
```

This assignment operator is only valid if the OtherExecutor type is convertible to Executor.

5.243.9.3 strand::operator= (3 of 4 overloads)

Move assignment operator.

```
strand & operator=(  
    strand && other);
```

5.243.9.4 strand::operator= (4 of 4 overloads)

Converting move assignment operator.

```
template<
    class OtherExecutor>
strand & operator=
    const strand< OtherExecutor > && other);
```

This assignment operator is only valid if the `OtherExecutor` type is convertible to `Executor`.

5.243.10 strand::operator==

Compare two strands for equality.

```
friend bool operator==
    const strand & a,
    const strand & b);
```

Two strands are equal if they refer to the same ordered, non-concurrent state.

Requirements

Header: `asio/strand.hpp`

Convenience header: `asio.hpp`

5.243.11 strand::post

Request the strand to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void post(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the executor to execute the given function object. The function object will never be executed inside this function. Instead, it will be scheduled by the underlying executor's `defer` function.

Parameters

- f** The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a** An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.243.12 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 function that was submitted to the strand using post(), dispatch() or defer(). Otherwise returns false.

5.243.13 strand::strand

Default constructor.

```
strand();
```

Construct a strand for the specified executor.

```
explicit strand(
    const Executor & e);
```

Copy constructor.

```
strand(
    const strand & other);
```

Converting constructor.

```
template<
    class OtherExecutor>
strand(
    const strand< OtherExecutor > & other);
```

Move constructor.

```
strand(
    strand && other);
```

Converting move constructor.

```
template<
    class OtherExecutor>
strand(
    strand< OtherExecutor > && other);
```

5.243.13.1 strand::strand (1 of 6 overloads)

Default constructor.

```
strand();
```

This constructor is only valid if the underlying executor type is default constructible.

5.243.13.2 strand::strand (2 of 6 overloads)

Construct a strand for the specified executor.

```
strand(
    const Executor & e);
```

5.243.13.3 strand::strand (3 of 6 overloads)

Copy constructor.

```
strand(  
    const strand & other);
```

5.243.13.4 strand::strand (4 of 6 overloads)

Converting constructor.

```
template<  
    class OtherExecutor>  
strand(  
    const strand< OtherExecutor > & other);
```

This constructor is only valid if the `OtherExecutor` type is convertible to `Executor`.

5.243.13.5 strand::strand (5 of 6 overloads)

Move constructor.

```
strand(  
    strand && other);
```

5.243.13.6 strand::strand (6 of 6 overloads)

Converting move constructor.

```
template<  
    class OtherExecutor>  
strand(  
    strand< OtherExecutor > && other);
```

This constructor is only valid if the `OtherExecutor` type is convertible to `Executor`.

5.243.14 strand::~strand

Destructor.

```
~strand();
```

5.244 streambuf

Typedef for the typical usage of `basic_streambuf`.

```
typedef basic_streambuf streambuf;
```

Types

Name	Description
<code>const_buffers_type</code>	The type used to represent the input sequence as a list of buffers.
<code>mutable_buffers_type</code>	The type used to represent the output sequence as a list of buffers. 133

Member Functions

Name	Description
<code>basic_streambuf</code>	Construct a <code>basic_streambuf</code> object.
<code>capacity</code>	Get the current capacity of the <code>basic_streambuf</code> .
<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`

5.245 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`

5.246 system_context

The executor context for the system executor.

```
class system_context :
    public execution_context
```

Types

Name	Description
executor_type	The executor type associated with the context.
fork_event	Fork-related event notifications.

Member Functions

Name	Description
get_executor	Obtain an executor for the context.
join	Join all threads in the system thread pool.
notify_fork	Notify the execution_context of a fork-related event.
stop	Signal all threads in the system thread pool to stop.
stopped	Determine whether the system thread pool has been stopped.
<code>~system_context</code>	Destructor shuts down all threads in the system thread pool.

Protected Member Functions

Name	Description
destroy	Destroys all services in the context.
shutdown	Shuts down all services in the context.

Friends

Name	Description
add_service	(Deprecated: Use make_service().) Add a service object to the execution_context.
has_service	Determine if an execution_context contains a specified service type.
make_service	Creates a service object and adds it to the execution_context.
use_service	Obtain the service object corresponding to the given type.

Requirements

Header: asio/system_context.hpp

Convenience header: asio.hpp

5.246.1 system_context::add_service

Inherited from execution_context.

(Deprecated: Use `make_service()`.) Add a service object to the `execution_context`.

```
template<
    typename Service>
friend void add_service(
    execution_context & e,
    Service * svc);
```

This function is used to add a service to the `execution_context`.

Parameters

`e` The `execution_context` object that owns the service.

`svc` The service object. On success, ownership of the service object is transferred to the `execution_context`. When the `execution_context` object is destroyed, it will destroy the service object by performing:

```
delete static_cast<execution_context::service*>(svc)
```

Exceptions

`asio::service_already_exists` Thrown if a service of the given type is already present in the `execution_context`.

`asio::invalid_service_owner` Thrown if the service's owning `execution_context` is not the `execution_context` object specified by the `e` parameter.

Requirements

Header: `asio/system_context.hpp`

Convenience header: `asio.hpp`

5.246.2 system_context::destroy

Inherited from execution_context.

Destroys all services in the context.

```
void destroy();
```

This function is implemented as follows:

- For each service object `svc` in the `execution_context` set, in reverse order * of the beginning of service object lifetime, performs
`delete static_cast<execution_context::service*>(svc).`

5.246.3 system_context::executor_type

The executor type associated with the context.

```
typedef system_executor executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the system executor to invoke the given function object.
dispatch	Request the system executor to invoke the given function object.
on_work_finished	Inform the executor that some work is no longer outstanding.
on_work_started	Inform the executor that it has some outstanding work to do.
post	Request the system executor to invoke the given function object.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

The system executor represents an execution context where functions are permitted to run on arbitrary threads. The `post()` and `defer()` functions schedule the function to run on an unspecified system thread pool, and `dispatch()` invokes the function immediately.

Requirements

Header: `asio/system_context.hpp`

Convenience header: `asio.hpp`

5.246.4 system_context::fork_event

Inherited from execution_context.

Fork-related event notifications.

```
enum fork_event
```

Values

fork_prepare Notify the context that the process is about to fork.

fork_parent Notify the context that the process has forked and is the parent.

fork_child Notify the context that the process has forked and is the child.

5.246.5 system_context::get_executor

Obtain an executor for the context.

```
executor_type get_executor();
```

5.246.6 system_context::has_service

Inherited from execution_context.

Determine if an [execution_context](#) contains a specified service type.

```
template<
    typename Service>
friend bool has_service(
    execution_context & e);
```

This function is used to determine whether the [execution_context](#) contains a service object corresponding to the given service type.

Parameters

e The [execution_context](#) object that owns the service.

Return Value

A boolean indicating whether the [execution_context](#) contains the service.

Requirements

Header: asio/system_context.hpp

Convenience header: asio.hpp

5.246.7 system_context::join

Join all threads in the system thread pool.

```
void join();
```

5.246.8 system_context::make_service

Inherited from execution_context.

Creates a service object and adds it to the [execution_context](#).

```
template<
    typename Service,
    typename... Args>
friend Service & make_service(
    execution_context & e,
    Args &&... args);
```

This function is used to add a service to the [execution_context](#).

Parameters

- **e** The `execution_context` object that owns the service.
- **args** Zero or more arguments to be passed to the service constructor.

Exceptions

`asio::service_already_exists` Thrown if a service of the given type is already present in the `execution_context`.

Requirements

Header: `asio/system_context.hpp`

Convenience header: `asio.hpp`

5.246.9 `system_context::notify_fork`

Inherited from execution_context.

Notify the `execution_context` of a fork-related event.

```
void notify_fork(  
    fork_event event);
```

This function is used to inform the `execution_context` that the process is about to fork, or has just forked. This allows the `execution_context`, 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 `execution_context` function, or any function associated with the `execution_context`'s derived class, 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 `execution_context` or its derived class.

Parameters

event A fork-related event.

Exceptions

`asio::system_error` Thrown on failure. If the notification fails the `execution_context` object should no longer be used and should be destroyed.

Example

The following code illustrates how to incorporate the `notify_fork()` function:

```
my_execution_context.notify_fork(execution_context::fork_prepare);  
if (fork() == 0)  
{  
    // This is the child process.  
    my_execution_context.notify_fork(execution_context::fork_child);  
}  
else  
{  
    // This is the parent process.  
    my_execution_context.notify_fork(execution_context::fork_parent);  
}
```

Remarks

For each service object `svc` in the `execution_context` set, performs `svc->notify_fork()`. 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.

5.246.10 system_context::shutdown

Inherited from execution_context.

Shuts down all services in the context.

```
void shutdown();
```

This function is implemented as follows:

- For each service object `svc` in the `execution_context` set, in reverse order of the beginning of service object lifetime, performs `svc->shutdown()`.

5.246.11 system_context::stop

Signal all threads in the system thread pool to stop.

```
void stop();
```

5.246.12 system_context::stopped

Determine whether the system thread pool has been stopped.

```
bool stopped() const;
```

5.246.13 system_context::use_service

Obtain the service object corresponding to the given type.

```
template<
    typename Service>
friend Service & use_service(
    execution_context & e);

template<
    typename Service>
friend Service & use_service(
    io_context & ioc);
```

5.246.13.1 system_context::use_service (1 of 2 overloads)

Inherited from execution_context.

Obtain the service object corresponding to the given type.

```
template<
    typename Service>
friend Service & use_service(
    execution_context & e);
```

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 `execution_context` will create a new instance of the service.

Parameters

- e The `execution_context` 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/system_context.hpp`

Convenience header: `asio.hpp`

5.246.13.2 `system_context::use_service` (2 of 2 overloads)

Inherited from execution_context.

Obtain the service object corresponding to the given type.

```
template<
    typename Service>
friend Service & use_service(
    io_context & ioc);
```

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_context` will create a new instance of the service.

Parameters

- ioc The `io_context` 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.

Remarks

This overload is preserved for backwards compatibility with services that inherit from `io_context::service`.

Requirements

Header: `asio/system_context.hpp`

Convenience header: `asio.hpp`

5.246.14 `system_context::~system_context`

Destructor shuts down all threads in the system thread pool.

```
~system_context();
```

5.247 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>code</code>	Get the error code associated with the exception.
<code>operator=</code>	Assignment operator.
<code>system_error</code>	Construct with an error code. Construct with an error code and context. Copy constructor.
<code>what</code>	Get a string representation of the exception.
<code>~system_error</code>	Destructor.

Requirements

Header: `asio/system_error.hpp`

Convenience header: `asio.hpp`

5.247.1 system_error::code

Get the error code associated with the exception.

```
error_code code() const;
```

5.247.2 system_error::operator=

Assignment operator.

```
system_error & operator=(  
    const system_error & e);
```

5.247.3 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);
```

5.247.3.1 system_error::system_error (1 of 3 overloads)

Construct with an error code.

```
system_error(
    const error_code & ec);
```

5.247.3.2 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);
```

5.247.3.3 system_error::system_error (3 of 3 overloads)

Copy constructor.

```
system_error(
    const system_error & other);
```

5.247.4 system_error::what

Get a string representation of the exception.

```
virtual const char * what() const;
```

5.247.5 system_error::~system_error

Destructor.

```
virtual ~system_error();
```

5.248 system_executor

An executor that uses arbitrary threads.

```
class system_executor
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the system executor to invoke the given function object.
dispatch	Request the system executor to invoke the given function object.
on_work_finished	Inform the executor that some work is no longer outstanding.
on_work_started	Inform the executor that it has some outstanding work to do.
post	Request the system executor to invoke the given function object.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

The system executor represents an execution context where functions are permitted to run on arbitrary threads. The `post()` and `defer()` functions schedule the function to run on an unspecified system thread pool, and `dispatch()` invokes the function immediately.

Requirements

Header: `asio/system_executor.hpp`

Convenience header: `asio.hpp`

5.248.1 system_executor::context

Obtain the underlying execution context.

```
system_context & context() const;
```

5.248.2 system_executor::defer

Request the system executor to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void defer(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the executor to execute the given function object. The function object will never be executed inside this function. Instead, it will be scheduled to run on an unspecified system thread pool.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.248.3 system_executor::dispatch

Request the system executor to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void dispatch(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the executor to execute the given function object. The function object will always be executed inside this function.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.248.4 system_executor::on_work_finished

Inform the executor that some work is no longer outstanding.

```
void on_work_finished() const;
```

For the system executor, this is a no-op.

5.248.5 system_executor::on_work_started

Inform the executor that it has some outstanding work to do.

```
void on_work_started() const;
```

For the system executor, this is a no-op.

5.248.6 system_executor::operator!=

Compare two executors for inequality.

```
friend bool operator!=(
    const system_executor & ,
    const system_executor & );
```

System executors always compare equal.

Requirements

Header: asio/system_executor.hpp

Convenience header: asio.hpp

5.248.7 system_executor::operator==

Compare two executors for equality.

```
friend bool operator==
    const system_executor &,
    const system_executor &);
```

System executors always compare equal.

Requirements

Header: asio/system_executor.hpp

Convenience header: asio.hpp

5.248.8 system_executor::post

Request the system executor to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void post(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the executor to execute the given function object. The function object will never be executed inside this function. Instead, it will be scheduled to run on an unspecified system thread pool.

Parameters

- f** The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a** An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.249 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.
executor_type	The type of the executor associated with the object.
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. Move-construct a basic_waitable_timer from another.
cancel	Cancel any asynchronous operations that are waiting on the timer. (Deprecated: Use non-error_code overload.) Cancel any asynchronous operations that are waiting on the timer.
cancel_one	Cancels one asynchronous operation that is waiting on the timer. (Deprecated: Use non-error_code overload.) Cancels one asynchronous operation that is waiting on the timer.
expires_after	Set the timer's expiry time relative to now.
expires_at	(Deprecated: Use expiry().) Get the timer's expiry time as an absolute time. Set the timer's expiry time as an absolute time. (Deprecated: Use non-error_code overload.) Set the timer's expiry time as an absolute time.
expires_from_now	(Deprecated: Use expiry().) Get the timer's expiry time relative to now. (Deprecated: Use expires_after().) Set the timer's expiry time relative to now.
expiry	Get the timer's expiry time as an absolute time.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.

Name	Description
<code>get_io_service</code>	(Deprecated: Use <code>get_executor()</code> .) Get the <code>io_context</code> associated with the object.
<code>operator=</code>	Move-assign a <code>basic_waitable_timer</code> from another.
<code>wait</code>	Perform a blocking wait on the timer.
<code>~basic_waitable_timer</code>	Destroys the timer.

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_context);

// Set an expiry time relative to now.
timer.expires_after(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_context,
    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_after(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_after()` 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: `asio.hpp`

5.250 thread

A simple abstraction for starting threads.

```
class thread :
    noncopyable
```

Member Functions

Name	Description
join	Wait for the thread to exit.

Name	Description
thread	Start a new thread that executes the supplied function.
~thread	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_context's event processing loop:

```
asio::io_context io_context;
// ...
asio::thread t(boost::bind(&asio::io_context::run, &io_context));
// ...
t.join();
```

Requirements

Header: asio/thread.hpp

Convenience header: asio.hpp

5.250.1 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.

5.250.2 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();
```

5.250.3 `thread::~thread`

Destructor.

```
~thread();
```

5.251 `thread_pool`

A simple fixed-size thread pool.

```
class thread_pool :  
    public execution_context
```

Types

Name	Description
<code>executor_type</code>	Executor used to submit functions to a thread pool.
<code>fork_event</code>	Fork-related event notifications.

Member Functions

Name	Description
<code>get_executor</code>	Obtains the executor associated with the pool.
<code>join</code>	Joins the threads.
<code>notify_fork</code>	Notify the <code>execution_context</code> of a fork-related event.
<code>stop</code>	Stops the threads.
<code>thread_pool</code>	Constructs a pool with an automatically determined number of threads. Constructs a pool with a specified number of threads.
<code>~thread_pool</code>	Destructor.

Protected Member Functions

Name	Description
destroy	Destroys all services in the context.
shutdown	Shuts down all services in the context.

Friends

Name	Description
add_service	(Deprecated: Use make_service().) Add a service object to the execution_context.
has_service	Determine if an execution_context contains a specified service type.
make_service	Creates a service object and adds it to the execution_context.
use_service	Obtain the service object corresponding to the given type.

The thread pool class is an execution context where functions are permitted to run on one of a fixed number of threads.

Submitting tasks to the pool

To submit functions to the `io_context`, use the `dispatch`, `post` or `defer` free functions.

For example:

```
void my_task()
{
    ...
}

// Launch the pool with four threads.
asio::thread_pool pool(4);

// Submit a function to the pool.
asio::post(pool, my_task);

// Submit a lambda object to the pool.
asio::post(pool,
    []()
    {
        ...
    });
}

// Wait for all tasks in the pool to complete.
pool.join();
```

Requirements

Header: `asio/thread_pool.hpp`

Convenience header: `asio.hpp`

5.251.1 `thread_pool::add_service`

Inherited from execution_context.

(Deprecated: Use `make_service()`.) Add a service object to the `execution_context`.

```
template<
    typename Service>
friend void add_service(
    execution_context & e,
    Service * svc);
```

This function is used to add a service to the `execution_context`.

Parameters

`e` The `execution_context` object that owns the service.

`svc` The service object. On success, ownership of the service object is transferred to the `execution_context`. When the `execution_context` object is destroyed, it will destroy the service object by performing:

```
delete static_cast<execution_context::service*>(svc)
```

Exceptions

`asio::service_already_exists` Thrown if a service of the given type is already present in the `execution_context`.

`asio::invalid_service_owner` Thrown if the service's owning `execution_context` is not the `execution_context` object specified by the `e` parameter.

Requirements

Header: `asio/thread_pool.hpp`

Convenience header: `asio.hpp`

5.251.2 `thread_pool::destroy`

Inherited from execution_context.

Destroys all services in the context.

```
void destroy();
```

This function is implemented as follows:

- For each service object `svc` in the `execution_context` set, in reverse order * of the beginning of service object lifetime, performs
`delete static_cast<execution_context::service*>(svc).`

5.251.3 `thread_pool::fork_event`

Inherited from execution_context.

Fork-related event notifications.

```
enum fork_event
```

Values

fork_prepare Notify the context that the process is about to fork.

fork_parent Notify the context that the process has forked and is the parent.

fork_child Notify the context that the process has forked and is the child.

5.251.4 `thread_pool::get_executor`

Obtains the executor associated with the pool.

```
executor_type get_executor();
```

5.251.5 `thread_pool::has_service`

Inherited from execution_context.

Determine if an `execution_context` contains a specified service type.

```
template<
    typename Service>
friend bool has_service(
    execution_context & e);
```

This function is used to determine whether the `execution_context` contains a service object corresponding to the given service type.

Parameters

`e` The `execution_context` object that owns the service.

Return Value

A boolean indicating whether the `execution_context` contains the service.

Requirements

Header: asio/thread_pool.hpp

Convenience header: asio.hpp

5.251.6 `thread_pool::join`

Joins the threads.

```
void join();
```

This function blocks until the threads in the pool have completed. If `stop()` is not called prior to `join()`, the `join()` call will wait until the pool has no more outstanding work.

5.251.7 `thread_pool::make_service`

Inherited from execution_context.

Creates a service object and adds it to the `execution_context`.

```
template<
    typename Service,
    typename... Args>
friend Service & make_service(
    execution_context & e,
    Args &&... args);
```

This function is used to add a service to the `execution_context`.

Parameters

e The `execution_context` object that owns the service.

args Zero or more arguments to be passed to the service constructor.

Exceptions

`asio::service_already_exists` Thrown if a service of the given type is already present in the `execution_context`.

Requirements

Header: `asio/thread_pool.hpp`

Convenience header: `asio.hpp`

5.251.8 `thread_pool::notify_fork`

Inherited from execution_context.

Notify the `execution_context` of a fork-related event.

```
void notify_fork(
    fork_event event);
```

This function is used to inform the `execution_context` that the process is about to fork, or has just forked. This allows the `execution_context`, 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 `execution_context` function, or any function associated with the `execution_context`'s derived class, 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 `execution_context` or its derived class.

Parameters

event A fork-related event.

Exceptions

`asio::system_error` Thrown on failure. If the notification fails the `execution_context` object should no longer be used and should be destroyed.

Example

The following code illustrates how to incorporate the `notify_fork()` function:

```
my_execution_context.notify_fork(execution_context::fork_prepare);
if (fork() == 0)
{
    // This is the child process.
    my_execution_context.notify_fork(execution_context::fork_child);
}
else
{
    // This is the parent process.
    my_execution_context.notify_fork(execution_context::fork_parent);
}
```

Remarks

For each service object `svc` in the `execution_context` set, performs `svc->notify_fork();`. 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.

5.251.9 `thread_pool::shutdown`

Inherited from execution_context.

Shuts down all services in the context.

```
void shutdown();
```

This function is implemented as follows:

- For each service object `svc` in the `execution_context` set, in reverse order of the beginning of service object lifetime, performs `svc->shutdown();`.

5.251.10 `thread_pool::stop`

Stops the threads.

```
void stop();
```

This function stops the threads as soon as possible. As a result of calling `stop()`, pending function objects may be never be invoked.

5.251.11 `thread_pool::thread_pool`

Constructs a pool with an automatically determined number of threads.

```
thread_pool();
```

Constructs a pool with a specified number of threads.

```
thread_pool(
    std::size_t num_threads);
```

5.251.11.1 `thread_pool::thread_pool (1 of 2 overloads)`

Constructs a pool with an automatically determined number of threads.

```
thread_pool();
```

5.251.11.2 `thread_pool::thread_pool (2 of 2 overloads)`

Constructs a pool with a specified number of threads.

```
thread_pool(  
    std::size_t num_threads);
```

5.251.12 `thread_pool::use_service`

Obtain the service object corresponding to the given type.

```
template<  
    typename Service>  
friend Service & use_service(  
    execution_context & e);  
  
template<  
    typename Service>  
friend Service & use_service(  
    io_context & ioc);
```

5.251.12.1 `thread_pool::use_service (1 of 2 overloads)`

Inherited from `execution_context`.

Obtain the service object corresponding to the given type.

```
template<  
    typename Service>  
friend Service & use_service(  
    execution_context & e);
```

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 `execution_context` will create a new instance of the service.

Parameters

e The `execution_context` 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/thread_pool.hpp`

Convenience header: `asio.hpp`

5.251.12.2 `thread_pool::use_service` (2 of 2 overloads)

Inherited from `execution_context`.

Obtain the service object corresponding to the given type.

```
template<
    typename Service>
friend Service & use_service(
    io_context & ioc);
```

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_context` will create a new instance of the service.

Parameters

`ioc` The `io_context` 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.

Remarks

This overload is preserved for backwards compatibility with services that inherit from `io_context::service`.

Requirements

Header: `asio/thread_pool.hpp`

Convenience header: `asio.hpp`

5.251.13 `thread_pool::~thread_pool`

Destructor.

```
~thread_pool();
```

Automatically stops and joins the pool, if not explicitly done beforehand.

5.252 `thread_pool::executor_type`

Executor used to submit functions to a thread pool.

```
class executor_type
```

Name	Description
------	-------------

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the thread pool to invoke the given function object.
dispatch	Request the thread pool to invoke the given function object.
on_work_finished	Inform the thread pool that some work is no longer outstanding.
on_work_started	Inform the thread pool that it has some outstanding work to do.
post	Request the thread pool to invoke the given function object.
running_in_this_thread	Determine whether the thread pool is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/thread_pool.hpp

Convenience header: asio.hpp

5.252.1 thread_pool::executor_type::context

Obtain the underlying execution context.

```
thread_pool & context() const;
```

5.252.2 thread_pool::executor_type::defer

Request the thread pool to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
```

```
void defer(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the thread pool to execute the given function object. The function object will never be executed inside `defer()`. Instead, it will be scheduled to run on the thread pool.

If the current thread belongs to the thread pool, `defer()` will delay scheduling the function object until the current thread returns control to the pool.

Parameters

- f** The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a** An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.252.3 `thread_pool::executor_type::dispatch`

Request the thread pool to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void dispatch(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the thread pool to execute the given function object. If the current thread belongs to the pool, `dispatch()` executes the function before returning. Otherwise, the function will be scheduled to run on the thread pool.

Parameters

- f** The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a** An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.252.4 `thread_pool::executor_type::on_work_finished`

Inform the thread pool that some work is no longer outstanding.

```
void on_work_finished() const;
```

This function is used to inform the thread pool that some work has finished. Once the count of unfinished work reaches zero, the thread pool's `join()` function is permitted to exit.

5.252.5 `thread_pool::executor_type::on_work_started`

Inform the thread pool that it has some outstanding work to do.

```
void on_work_started() const;
```

This function is used to inform the thread pool that some work has begun. This ensures that the thread pool's `join()` function will not return while the work is underway.

5.252.6 `thread_pool::executor_type::operator!=`

Compare two executors for inequality.

```
friend bool operator!=(
    const executor_type & a,
    const executor_type & b);
```

Two executors are equal if they refer to the same underlying thread pool.

Requirements

Header: `asio/thread_pool.hpp`

Convenience header: `asio.hpp`

5.252.7 `thread_pool::executor_type::operator==`

Compare two executors for equality.

```
friend bool operator==((
    const executor_type & a,
    const executor_type & b);
```

Two executors are equal if they refer to the same underlying thread pool.

Requirements

Header: `asio/thread_pool.hpp`

Convenience header: `asio.hpp`

5.252.8 `thread_pool::executor_type::post`

Request the thread pool to invoke the given function object.

```
template<
    typename Function,
    typename Allocator>
void post(
    Function && f,
    const Allocator & a) const;
```

This function is used to ask the thread pool to execute the given function object. The function object will never be executed inside `post()`. Instead, it will be scheduled to run on the thread pool.

Parameters

- f The function object to be called. The executor will make a copy of the handler object as required. The function signature of the function object must be:

```
void function();
```

- a An allocator that may be used by the executor to allocate the internal storage needed for function invocation.

5.252.9 `thread_pool::executor_type::running_in_this_thread`

Determine whether the thread pool is running in the current thread.

```
bool running_in_this_thread() const;
```

Return Value

`true` if the current thread belongs to the pool. Otherwise returns `false`.

5.253 `time_traits< boost::posix_time::ptime >`

Time traits specialised for `posix_time`.

```
template<>
struct time_traits< boost::posix_time::ptime >
```

Types

Name	Description
<code>duration_type</code>	The duration type.
<code>time_type</code>	The time type.

Member Functions

Name	Description
<code>add</code>	Add a duration to a time.
<code>less_than</code>	Test whether one time is less than another.
<code>now</code>	Get the current time.
<code>subtract</code>	Subtract one time from another.
<code>to_posix_duration</code>	Convert to POSIX duration type.

Requirements

Header: asio/time_traits.hpp

Convenience header: asio.hpp

5.253.1 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);
```

5.253.2 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

5.253.3 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);
```

5.253.4 time_traits< boost::posix_time::ptime >::now

Get the current time.

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

5.253.5 time_traits< boost::posix_time::ptime >::subtract

Subtract one time from another.

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

5.253.6 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

5.253.7 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);
```

5.254 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

5.255 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

5.256 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

5.257 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: asio.hpp

5.258 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()	Wrap a function object in a packaged task.
operator[]	(Deprecated: Use rebind() .) Specify an alternate allocator.
rebind	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: asio.hpp

5.258.1 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: asio.hpp

5.258.2 use_future_t::get_allocator

Obtain allocator.

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

5.258.3 use_future_t::operator()

Wrap a function object in a packaged task.

```
template<
    typename Function>
unspecified operator()(
    Function && f) const;
```

The `package` function is used to adapt a function object as a packaged task. When this adapter is passed as a completion token to an asynchronous operation, the result of the function object is returned via a `std::future`.

Example

```
std::future<std::size_t> fut =
    my_socket.async_read_some(buffer,
        use_future([](asio::error_code ec, std::size_t n)
    {
        return ec ? 0 : n;
    }));
...
std::size_t n = fut.get();
```

5.258.4 `use_future_t::operator[]`

(Deprecated: Use `rebind()`.) Specify an alternate allocator.

```
template<
    typename OtherAllocator>
use_future_t< OtherAllocator > operator[] (
    const OtherAllocator & allocator) const;
```

5.258.5 `use_future_t::rebind`

Specify an alternate allocator.

```
template<
    typename OtherAllocator>
use_future_t< OtherAllocator > rebind(
    const OtherAllocator & allocator) const;
```

5.258.6 `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);
```

5.258.6.1 `use_future_t::use_future_t (1 of 2 overloads)`

Construct using default-constructed allocator.

```
constexpr use_future_t();
```

5.258.6.2 `use_future_t::use_future_t (2 of 2 overloads)`

Construct using specified allocator.

```
use_future_t(
    const Allocator & allocator);
```

5.259 `use_service`

```
template<
    typename Service>
Service & use_service(
    execution_context & e);
```

```
template<
    typename Service>
Service & use_service(
    io_context & ioc);
```

Requirements

Header: asio/impl/execution_context.hpp

Convenience header: asio.hpp

5.259.1 use_service (1 of 2 overloads)

```
template<
    typename Service>
Service & use_service(
    execution_context & e);
```

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 [execution_context](#) will create a new instance of the service.

Parameters

e The [execution_context](#) 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.

5.259.2 use_service (2 of 2 overloads)

```
template<
    typename Service>
Service & use_service(
    io_context & ioc);
```

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_context](#) will create a new instance of the service.

Parameters

ioc The [io_context](#) 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.

Remarks

This overload is preserved for backwards compatibility with services that inherit from [io_context::service](#).

5.260 uses_executor

The `uses_executor` trait detects whether a type T has an associated executor that is convertible from type Executor.

```
template<
    typename T,
    typename Executor>
struct uses_executor
```

Meets the BinaryTypeTrait requirements. The Asio library provides a definition that is derived from `false_type`. A program may specialize this template to derive from `true_type` for a user-defined type T that can be constructed with an executor, where the first argument of a constructor has type `executor_arg_t` and the second argument is convertible from type Executor.

Requirements

Header: asio/uses_executor.hpp

Convenience header: asio.hpp

5.261 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

5.261.1 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);

static Clock::duration to_wait_duration(
    const typename Clock::time_point & t);
```

5.261.1.1 wait_traits::to_wait_duration (1 of 2 overloads)

Convert a clock duration into a duration used for waiting.

```
static Clock::duration to_wait_duration(
    const typename Clock::duration & d);
```

Return Value

d.

5.261.1.2 wait_traits::to_wait_duration (2 of 2 overloads)

Convert a clock duration into a duration used for waiting.

```
static Clock::duration to_wait_duration(
    const typename Clock::time_point & t);
```

Return Value

d.

5.262 windows::object_handle

Provides object-oriented handle functionality.

```
class object_handle
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
lowest_layer_type	An object_handle is always the lowest layer.
native_handle_type	The native representation of a handle.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
async_wait	Start an asynchronous wait on the object handle.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_executor	Get the executor associated with the object.

Name	Description
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_handle	Get the native handle representation.
object_handle	Construct an object_handle without opening it. Construct an object_handle on an existing native handle. Move-construct an object_handle from another.
operator=	Move-assign an object_handle from another.
wait	Perform a blocking wait on the object handle.

The `windows::object_handle` class 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`

5.262.1 windows::object_handle::assign

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);

void assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.262.1.1 windows::object_handle::assign (1 of 2 overloads)

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);
```

5.262.1.2 windows::object_handle::assign (2 of 2 overloads)

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.262.2 windows::object_handle::async_wait

Start an asynchronous wait on the object handle.

```
template<
    typename WaitHandler>
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_context::post()`.

5.262.3 windows::object_handle::cancel

Cancel all asynchronous operations associated with the handle.

```
void cancel();

void cancel(
    asio::error_code & ec);
```

5.262.3.1 windows::object_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.

5.262.3.2 windows::object_handle::cancel (2 of 2 overloads)

Cancel all asynchronous operations associated with the handle.

```
void 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.

5.262.4 windows::object_handle::close

Close the handle.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.262.4.1 windows::object_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.

5.262.4.2 windows::object_handle::close (2 of 2 overloads)

Close the handle.

```
void 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.

5.262.5 windows::object_handle::executor_type

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the io_context to invoke the given function object.
dispatch	Request the io_context to invoke the given function object.
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/windows/object_handle.hpp

Convenience header: asio.hpp

5.262.6 windows::object_handle::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.262.7 windows::object_handle::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the [io_context](#) object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the [io_context](#) object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.262.8 windows::object_handle::get_io_service

(Deprecated: Use `get_executor()`.) Get the [io_context](#) associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the [io_context](#) object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the [io_context](#) object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.262.9 windows::object_handle::is_open

Determine whether the handle is open.

```
bool is_open() const;
```

5.262.10 windows::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;
```

5.262.10.1 windows::object_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 an [windows::object_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.

5.262.10.2 windows::object_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 an `windows::object_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.

5.262.11 windows::object_handle::lowest_layer_type

An `windows::object_handle` is always the lowest layer.

```
typedef object_handle lowest_layer_type;
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
lowest_layer_type	An <code>object_handle</code> is always the lowest layer.
native_handle_type	The native representation of a handle.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
async_wait	Start an asynchronous wait on the object handle.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use <code>get_executor()</code> .) Get the <code>io_context</code> associated with the object.
get_io_service	(Deprecated: Use <code>get_executor()</code> .) Get the <code>io_context</code> 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_handle	Get the native handle representation.
object_handle	Construct an object_handle without opening it. Construct an object_handle on an existing native handle. Move-construct an object_handle from another.
operator=	Move-assign an object_handle from another.
wait	Perform a blocking wait on the object handle.

The `windows::object_handle` class 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`

5.262.12 windows::object_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.

5.262.13 windows::object_handle::native_handle_type

The native representation of a handle.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: `asio/windows/object_handle.hpp`

Convenience header: `asio.hpp`

5.262.14 windows::object_handle::object_handle

Construct an `windows::object_handle` without opening it.

```
explicit object_handle(
    asio::io_context & io_context);
```

Construct an `windows::object_handle` on an existing native handle.

```
object_handle(
    asio::io_context & io_context,
    const native_handle_type & native_handle);
```

Move-construct an `windows::object_handle` from another.

```
object_handle(
    object_handle && other);
```

5.262.14.1 windows::object_handle::object_handle (1 of 3 overloads)

Construct an `windows::object_handle` without opening it.

```
object_handle(
    asio::io_context & io_context);
```

This constructor creates an object handle without opening it.

Parameters

io_context The `io_context` object that the object handle will use to dispatch handlers for any asynchronous operations performed on the handle.

5.262.14.2 windows::object_handle::object_handle (2 of 3 overloads)

Construct an `windows::object_handle` on an existing native handle.

```
object_handle(
    asio::io_context & io_context,
    const native_handle_type & native_handle);
```

This constructor creates an object handle object to hold an existing native handle.

Parameters

io_context The `io_context` 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.

5.262.14.3 windows::object_handle::object_handle (3 of 3 overloads)

Move-construct an `windows::object_handle` from another.

```
object_handle(
    object_handle && other);
```

This constructor moves an object handle from one object to another.

Parameters

other The other `windows::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 `object_handle(io_context&)` constructor.

5.262.15 windows::object_handle::operator=

Move-assign an `windows::object_handle` from another.

```
object_handle & operator=(
    object_handle && other);
```

This assignment operator moves an object handle from one object to another.

Parameters

other The other `windows::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 `object_handle(io_context&)` constructor.

5.262.16 windows::object_handle::wait

Perform a blocking wait on the object handle.

```
void wait();

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

5.262.16.1 windows::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.

5.262.16.2 windows::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.

5.263 windows::overlapped_handle

Provides Windows handle functionality for objects that support overlapped I/O.

```
class overlapped_handle
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
lowest_layer_type	An overlapped_handle is always the lowest layer.
native_handle_type	The native representation of a handle.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.

Name	Description
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_handle	Get the native handle representation.
operator=	Move-assign an overlapped_handle from another.
overlapped_handle	Construct an overlapped_handle without opening it. Construct an overlapped_handle on an existing native handle. Move-construct an overlapped_handle from another.

Protected Member Functions

Name	Description
~overlapped_handle	Protected destructor to prevent deletion through this type.

The `windows::overlapped_handle` class provides the ability to wrap a Windows handle. The underlying object referred to by the handle must support overlapped I/O.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/overlapped_handle.hpp`

Convenience header: `asio.hpp`

5.263.1 windows::overlapped_handle::assign

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);

void assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.263.1.1 windows::overlapped_handle::assign (1 of 2 overloads)

Assign an existing native handle to the handle.

```
void assign(  
    const native_handle_type & handle);
```

5.263.1.2 windows::overlapped_handle::assign (2 of 2 overloads)

Assign an existing native handle to the handle.

```
void assign(  
    const native_handle_type & handle,  
    asio::error_code & ec);
```

5.263.2 windows::overlapped_handle::cancel

Cancel all asynchronous operations associated with the handle.

```
void cancel();  
  
void cancel(  
    asio::error_code & ec);
```

5.263.2.1 windows::overlapped_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.

5.263.2.2 windows::overlapped_handle::cancel (2 of 2 overloads)

Cancel all asynchronous operations associated with the handle.

```
void 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.

5.263.3 windows::overlapped_handle::close

Close the handle.

```
void close();  
  
void close(  
   asio::error_code & ec);
```

5.263.3.1 windows::overlapped_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.

5.263.3.2 windows::overlapped_handle::close (2 of 2 overloads)

Close the handle.

```
void 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.

5.263.4 windows::overlapped_handle::executor_type

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
context	Obtain the underlying execution context.
defer	Request the <code>io_context</code> to invoke the given function object.
dispatch	Request the <code>io_context</code> to invoke the given function object.

Name	Description
on_work_finished	Inform the io_context that some work is no longer outstanding.
on_work_started	Inform the io_context that it has some outstanding work to do.
post	Request the io_context to invoke the given function object.
running_in_this_thread	Determine whether the io_context is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/windows/overlapped_handle.hpp

Convenience header: asio.hpp

5.263.5 windows::overlapped_handle::get_executor

Get the executor associated with the object.

```
executor_type get_executor();
```

5.263.6 windows::overlapped_handle::get_io_context

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.263.7 windows::overlapped_handle::get_io_service

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.263.8 windows::overlapped_handle::is_open

Determine whether the handle is open.

```
bool is_open() const;
```

5.263.9 windows::overlapped_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;
```

5.263.9.1 windows::overlapped_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 an `windows::overlapped_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.

5.263.9.2 windows::overlapped_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 an `windows::overlapped_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.

5.263.10 windows::overlapped_handle::lowest_layer_type

An `windows::overlapped_handle` is always the lowest layer.

```
typedef overlapped_handle lowest_layer_type;
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
lowest_layer_type	An overlapped_handle is always the lowest layer.
native_handle_type	The native representation of a handle.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_handle	Get the native handle representation.
operator=	Move-assign an overlapped_handle from another.
overlapped_handle	Construct an overlapped_handle without opening it. Construct an overlapped_handle on an existing native handle. Move-construct an overlapped_handle from another.

Protected Member Functions

Name	Description
~overlapped_handle	Protected destructor to prevent deletion through this type.

The `windows::overlapped_handle` class provides the ability to wrap a Windows handle. The underlying object referred to by the handle must support overlapped I/O.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/windows/overlapped_handle.hpp

Convenience header: asio.hpp

5.263.11 windows::overlapped_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.

5.263.12 windows::overlapped_handle::native_handle_type

The native representation of a handle.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/windows/overlapped_handle.hpp

Convenience header: asio.hpp

5.263.13 windows::overlapped_handle::operator=

Move-assign an `windows::overlapped_handle` from another.

```
overlapped_handle & operator=(  
    overlapped_handle && other);
```

This assignment operator moves a handle from one object to another.

Parameters

other The other `windows::overlapped_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 `overlapped_handle(io_context &)` constructor.

5.263.14 windows::overlapped_handle::overlapped_handle

Construct an `windows::overlapped_handle` without opening it.

```
explicit overlapped_handle(
    asio::io_context & io_context);
```

Construct an `windows::overlapped_handle` on an existing native handle.

```
overlapped_handle(
    asio::io_context & io_context,
    const native_handle_type & handle);
```

Move-construct an `windows::overlapped_handle` from another.

```
overlapped_handle(
    overlapped_handle && other);
```

5.263.14.1 windows::overlapped_handle::overlapped_handle (1 of 3 overloads)

Construct an `windows::overlapped_handle` without opening it.

```
overlapped_handle(
    asio::io_context & io_context);
```

This constructor creates a handle without opening it.

Parameters

io_context The `io_context` object that the handle will use to dispatch handlers for any asynchronous operations performed on the handle.

5.263.14.2 windows::overlapped_handle::overlapped_handle (2 of 3 overloads)

Construct an `windows::overlapped_handle` on an existing native handle.

```
overlapped_handle(
    asio::io_context & io_context,
    const native_handle_type & handle);
```

This constructor creates a handle object to hold an existing native handle.

Parameters

io_context The `io_context` 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.

5.263.14.3 windows::overlapped_handle::overlapped_handle (3 of 3 overloads)

Move-construct an windows::overlapped_handle from another.

```
overlapped_handle(  
    overlapped_handle && other);
```

This constructor moves a handle from one object to another.

Parameters

other The other windows::overlapped_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 overlapped_handle(io_context&) constructor.

5.263.15 windows::overlapped_handle::~overlapped_handle

Protected destructor to prevent deletion through this type.

```
~overlapped_handle();
```

This function destroys the handle, cancelling any outstanding asynchronous wait operations associated with the handle as if by calling cancel.

5.264 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

5.264.1 windows::overlapped_ptr::complete

Post completion notification for overlapped operation. Releases ownership.

```
void complete(  
    const asio::error_code & ec,  
    std::size_t bytes_transferred);
```

5.264.2 windows::overlapped_ptr::get

Get the contained OVERLAPPED object.

```
OVERLAPPED * get();  
  
const OVERLAPPED * get() const;
```

5.264.2.1 windows::overlapped_ptr::get (1 of 2 overloads)

Get the contained OVERLAPPED object.

```
OVERLAPPED * get();
```

5.264.2.2 windows::overlapped_ptr::get (2 of 2 overloads)

Get the contained OVERLAPPED object.

```
const OVERLAPPED * get() const;
```

5.264.3 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_context & io_context,
    Handler && handler);
```

5.264.3.1 windows::overlapped_ptr::overlapped_ptr (1 of 2 overloads)

Construct an empty `windows::overlapped_ptr`.

```
overlapped_ptr();
```

5.264.3.2 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_context & io_context,
    Handler && handler);
```

5.264.4 windows::overlapped_ptr::release

Release ownership of the OVERLAPPED object.

```
OVERLAPPED * release();
```

5.264.5 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_context & io_context,
    Handler && handler);
```

5.264.5.1 windows::overlapped_ptr::reset (1 of 2 overloads)

Reset to empty.

```
void reset();
```

5.264.5.2 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_context & io_context,
    Handler && handler);
```

5.264.6 windows::overlapped_ptr::~overlapped_ptr

Destructor automatically frees the OVERLAPPED object unless released.

```
~overlapped_ptr();
```

5.265 windows::random_access_handle

Provides random-access handle functionality.

```
class random_access_handle :
    public windows::overlapped_handle
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
lowest_layer_type	An overlapped_handle is always the lowest layer.
native_handle_type	The native representation of a handle.

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.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.

Name	Description
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_handle	Get the native handle representation.
operator=	Move-assign a random_access_handle from another.
random_access_handle	Construct a random_access_handle without opening it. Construct a random_access_handle on an existing native handle. Move-construct a 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.

The `windows::random_access_handle` class 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`

5.265.1 windows::random_access_handle::assign

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);

void assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.265.1.1 windows::random_access_handle::assign (1 of 2 overloads)

Inherited from windows::overlapped_handle.

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle);
```

5.265.1.2 windows::random_access_handle::assign (2 of 2 overloads)

Inherited from windows::overlapped_handle.

Assign an existing native handle to the handle.

```
void assign(
    const native_handle_type & handle,
    asio::error_code & ec);
```

5.265.2 windows::random_access_handle::async_read_some_at

Start an asynchronous read at the specified offset.

```
template<
    typename MutableBufferSequence,
    typename ReadHandler>
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_context::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`.

5.265.3 windows::random_access_handle::async_write_some_at

Start an asynchronous write at the specified offset.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
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_context::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`.

5.265.4 windows::random_access_handle::cancel

Cancel all asynchronous operations associated with the handle.

```
void cancel();  
  
void cancel(  
    asio::error_code & ec);
```

5.265.4.1 windows::random_access_handle::cancel (1 of 2 overloads)

Inherited from windows::overlapped_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.

5.265.4.2 windows::random_access_handle::cancel (2 of 2 overloads)

Inherited from windows::overlapped_handle.

Cancel all asynchronous operations associated with the handle.

```
void 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.

5.265.5 windows::random_access_handle::close

Close the handle.

```
void close();
```

```
void close(  
    asio::error_code & ec);
```

5.265.5.1 windows::random_access_handle::close (1 of 2 overloads)

Inherited from windows::overlapped_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.

5.265.5.2 windows::random_access_handle::close (2 of 2 overloads)

Inherited from windows::overlapped_handle.

Close the handle.

```
void 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.

5.265.6 windows::random_access_handle::executor_type

Inherited from windows::overlapped_handle.

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
<code>context</code>	Obtain the underlying execution context.
<code>defer</code>	Request the <code>io_context</code> to invoke the given function object.
<code>dispatch</code>	Request the <code>io_context</code> to invoke the given function object.
<code>on_work_finished</code>	Inform the <code>io_context</code> that some work is no longer outstanding.
<code>on_work_started</code>	Inform the <code>io_context</code> that it has some outstanding work to do.
<code>post</code>	Request the <code>io_context</code> to invoke the given function object.
<code>running_in_this_thread</code>	Determine whether the <code>io_context</code> is running in the current thread.

Friends

Name	Description
<code>operator!=</code>	Compare two executors for inequality.
<code>operator==</code>	Compare two executors for equality.

Requirements

Header: asio/windows/random_access_handle.hpp

Convenience header: asio.hpp

5.265.7 windows::random_access_handle::get_executor

Inherited from windows::overlapped_handle.

Get the executor associated with the object.

```
executor_type get_executor();
```

5.265.8 windows::random_access_handle::get_io_context

Inherited from windows::overlapped_handle.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.265.9 windows::random_access_handle::get_io_service

Inherited from windows::overlapped_handle.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.265.10 windows::random_access_handle::is_open

Inherited from windows::overlapped_handle.

Determine whether the handle is open.

```
bool is_open() const;
```

5.265.11 windows::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;
```

5.265.11.1 windows::random_access_handle::lowest_layer (1 of 2 overloads)

Inherited from windows::overlapped_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 an [windows::overlapped_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.

5.265.11.2 windows::random_access_handle::lowest_layer (2 of 2 overloads)

Inherited from windows::overlapped_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 an [windows::overlapped_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.

5.265.12 windows::random_access_handle::lowest_layer_type

Inherited from windows::overlapped_handle.

An [windows::overlapped_handle](#) is always the lowest layer.

```
typedef overlapped_handle lowest_layer_type;
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
lowest_layer_type	An overlapped_handle is always the lowest layer.
native_handle_type	1405 The native representation of a handle.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_handle	Get the native handle representation.
operator=	Move-assign an overlapped_handle from another.
overlapped_handle	Construct an overlapped_handle without opening it. Construct an overlapped_handle on an existing native handle. Move-construct an overlapped_handle from another.

Protected Member Functions

Name	Description
~overlapped_handle	Protected destructor to prevent deletion through this type.

The `windows::overlapped_handle` class provides the ability to wrap a Windows handle. The underlying object referred to by the handle must support overlapped I/O.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: `asio/windows/random_access_handle.hpp`

Convenience header: `asio.hpp`

5.265.13 windows::random_access_handle::native_handle

Inherited from windows::overlapped_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.

5.265.14 windows::random_access_handle::native_handle_type

Inherited from windows::overlapped_handle.

The native representation of a handle.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/windows/random_access_handle.hpp

Convenience header: asio.hpp

5.265.15 windows::random_access_handle::operator=

Move-assign a windows::random_access_handle from another.

```
random_access_handle & operator=(  
    random_access_handle && other);
```

This assignment operator moves a random-access handle from one object to another.

Parameters

other The other windows::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 random_access_handle(io_context&) constructor.

5.265.16 windows::random_access_handle::random_access_handle

Construct a windows::random_access_handle without opening it.

```
explicit random_access_handle(  
    asio::io_context & io_context);
```

Construct a windows::random_access_handle on an existing native handle.

```
random_access_handle(
    asio::io_context & io_context,
    const native_handle_type & handle);
```

Move-construct a `windows::random_access_handle` from another.

```
random_access_handle(
    random_access_handle && other);
```

5.265.16.1 `windows::random_access_handle::random_access_handle (1 of 3 overloads)`

Construct a `windows::random_access_handle` without opening it.

```
random_access_handle(
    asio::io_context & io_context);
```

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_context The `io_context` object that the random-access handle will use to dispatch handlers for any asynchronous operations performed on the handle.

5.265.16.2 `windows::random_access_handle::random_access_handle (2 of 3 overloads)`

Construct a `windows::random_access_handle` on an existing native handle.

```
random_access_handle(
    asio::io_context & io_context,
    const native_handle_type & handle);
```

This constructor creates a random-access handle object to hold an existing native handle.

Parameters

io_context The `io_context` 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.

5.265.16.3 `windows::random_access_handle::random_access_handle (3 of 3 overloads)`

Move-construct a `windows::random_access_handle` from another.

```
random_access_handle(
    random_access_handle && other);
```

This constructor moves a random-access handle from one object to another.

Parameters

other The other `windows::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 `random_access_handle(io_context&)` constructor.

5.265.17 windows::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);
```

5.265.17.1 windows::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`.

5.265.17.2 windows::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.

5.265.18 windows::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);
```

5.265.18.1 windows::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`.

5.265.18.2 windows::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.

5.266 windows::stream_handle

Provides stream-oriented handle functionality.

```
class stream_handle :
    public windows::overlapped_handle
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
lowest_layer_type	An overlapped_handle is always the lowest layer.
native_handle_type	The native representation of a handle.

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.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_handle	Get the native handle representation.
operator=	Move-assign a stream_handle from another.
read_some	Read some data from the handle.
stream_handle	Construct a stream_handle without opening it. Construct a stream_handle on an existing native handle. Move-construct a stream_handle from another.
write_some	Write some data to the handle.

The `windows::stream_handle` class 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`

5.266.1 windows::stream_handle::assign

Assign an existing native handle to the handle.

```

void assign(
    const native_handle_type & handle);

void assign(
    const native_handle_type & handle,
    asio::error_code & ec);

```

5.266.1.1 windows::stream_handle::assign (1 of 2 overloads)

Inherited from windows::overlapped_handle.

Assign an existing native handle to the handle.

```

void assign(
    const native_handle_type & handle);

```

5.266.1.2 windows::stream_handle::assign (2 of 2 overloads)

Inherited from windows::overlapped_handle.

Assign an existing native handle to the handle.

```

void assign(
    const native_handle_type & handle,
    asio::error_code & ec);

```

5.266.2 windows::stream_handle::async_read_some

Start an asynchronous read.

```

template<
    typename MutableBufferSequence,
    typename ReadHandler>
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_context::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`.

5.266.3 windows::stream_handle::async_write_some

Start an asynchronous write.

```
template<
    typename ConstBufferSequence,
    typename WriteHandler>
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_context::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`.

5.266.4 windows::stream_handle::cancel

Cancel all asynchronous operations associated with the handle.

```
void cancel();  
  
void cancel(  
    asio::error_code & ec);
```

5.266.4.1 windows::stream_handle::cancel (1 of 2 overloads)

Inherited from windows::overlapped_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.

5.266.4.2 windows::stream_handle::cancel (2 of 2 overloads)

Inherited from windows::overlapped_handle.

Cancel all asynchronous operations associated with the handle.

```
void 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.

5.266.5 windows::stream_handle::close

Close the handle.

```
void close();  
  
void close(  
    asio::error_code & ec);
```

5.266.5.1 windows::stream_handle::close (1 of 2 overloads)

Inherited from windows::overlapped_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.

5.266.5.2 windows::stream_handle::close (2 of 2 overloads)

Inherited from windows::overlapped_handle.

Close the handle.

```
void 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.

5.266.6 windows::stream_handle::executor_type

Inherited from windows::overlapped_handle.

The type of the executor associated with the object.

```
typedef io_context::executor_type executor_type;
```

Member Functions

Name	Description
<code>context</code>	Obtain the underlying execution context.
<code>defer</code>	Request the <code>io_context</code> to invoke the given function object.
<code>dispatch</code>	Request the <code>io_context</code> to invoke the given function object.
<code>on_work_finished</code>	Inform the <code>io_context</code> that some work is no longer outstanding.
<code>on_work_started</code>	Inform the <code>io_context</code> that it has some outstanding work to do.
<code>post</code>	Request the <code>io_context</code> to invoke the given function object.
<code>running_in_this_thread</code>	Determine whether the <code>io_context</code> is running in the current thread.

Friends

Name	Description
operator!=	Compare two executors for inequality.
operator==	Compare two executors for equality.

Requirements

Header: asio/windows/stream_handle.hpp

Convenience header: asio.hpp

5.266.7 windows::stream_handle::get_executor

Inherited from windows::overlapped_handle.

Get the executor associated with the object.

```
executor_type get_executor();
```

5.266.8 windows::stream_handle::get_io_context

Inherited from windows::overlapped_handle.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_context();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.266.9 windows::stream_handle::get_io_service

Inherited from windows::overlapped_handle.

(Deprecated: Use `get_executor()`.) Get the `io_context` associated with the object.

```
asio::io_context & get_io_service();
```

This function may be used to obtain the `io_context` object that the I/O object uses to dispatch handlers for asynchronous operations.

Return Value

A reference to the `io_context` object that the I/O object will use to dispatch handlers. Ownership is not transferred to the caller.

5.266.10 windows::stream_handle::is_open

Inherited from windows::overlapped_handle.

Determine whether the handle is open.

```
bool is_open() const;
```

5.266.11 windows::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;
```

5.266.11.1 windows::stream_handle::lowest_layer (1 of 2 overloads)

Inherited from windows::overlapped_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 an [windows::overlapped_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.

5.266.11.2 windows::stream_handle::lowest_layer (2 of 2 overloads)

Inherited from windows::overlapped_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 an [windows::overlapped_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.

5.266.12 windows::stream_handle::lowest_layer_type

Inherited from windows::overlapped_handle.

An [windows::overlapped_handle](#) is always the lowest layer.

```
typedef overlapped_handle lowest_layer_type;
```

Types

Name	Description
executor_type	The type of the executor associated with the object.
lowest_layer_type	An overlapped_handle is always the lowest layer.
native_handle_type	The native representation of a handle.

Member Functions

Name	Description
assign	Assign an existing native handle to the handle.
cancel	Cancel all asynchronous operations associated with the handle.
close	Close the handle.
get_executor	Get the executor associated with the object.
get_io_context	(Deprecated: Use get_executor().) Get the io_context associated with the object.
get_io_service	(Deprecated: Use get_executor().) Get the io_context 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_handle	Get the native handle representation.
operator=	Move-assign an overlapped_handle from another.
overlapped_handle	Construct an overlapped_handle without opening it. Construct an overlapped_handle on an existing native handle. Move-construct an overlapped_handle from another.

Protected Member Functions

Name	Description
~overlapped_handle	Protected destructor to prevent deletion through this type.

The `windows::overlapped_handle` class provides the ability to wrap a Windows handle. The underlying object referred to by the handle must support overlapped I/O.

Thread Safety

Distinct objects: Safe.

Shared objects: Unsafe.

Requirements

Header: asio/windows/stream_handle.hpp

Convenience header: asio.hpp

5.266.13 windows::stream_handle::native_handle

Inherited from windows::overlapped_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.

5.266.14 windows::stream_handle::native_handle_type

Inherited from windows::overlapped_handle.

The native representation of a handle.

```
typedef implementation_defined native_handle_type;
```

Requirements

Header: asio/windows/stream_handle.hpp

Convenience header: asio.hpp

5.266.15 windows::stream_handle::operator=

Move-assign a `windows::stream_handle` from another.

```
stream_handle & operator=(  
    stream_handle && other);
```

This assignment operator moves a stream handle from one object to another.

Parameters

other The other `windows::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 `stream_handle(io_context &)` constructor.

5.266.16 windows::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);
```

5.266.16.1 windows::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`.

5.266.16.2 windows::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.

5.266.17 windows::stream_handle::stream_handle

Construct a [windows::stream_handle](#) without opening it.

```
explicit stream_handle(
    asio::io_context & io_context);
```

Construct a [windows::stream_handle](#) on an existing native handle.

```
stream_handle(
    asio::io_context & io_context,
    const native_handle_type & handle);
```

Move-construct a [windows::stream_handle](#) from another.

```
stream_handle(
    stream_handle && other);
```

5.266.17.1 windows::stream_handle::stream_handle (1 of 3 overloads)

Construct a [windows::stream_handle](#) without opening it.

```
stream_handle(
    asio::io_context & io_context);
```

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_context The `io_context` object that the stream handle will use to dispatch handlers for any asynchronous operations performed on the handle.

5.266.17.2 windows::stream_handle::stream_handle (2 of 3 overloads)

Construct a `windows::stream_handle` on an existing native handle.

```
stream_handle(
    asio::io_context & io_context,
    const native_handle_type & handle);
```

This constructor creates a stream handle object to hold an existing native handle.

Parameters

io_context The `io_context` 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.

5.266.17.3 windows::stream_handle::stream_handle (3 of 3 overloads)

Move-construct a `windows::stream_handle` from another.

```
stream_handle(
    stream_handle && other);
```

This constructor moves a stream handle from one object to another.

Parameters

other The other `windows::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 `stream_handle(io_context &)` constructor.

5.266.18 windows::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);
```

5.266.18.1 windows::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`.

5.266.18.2 windows::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.

5.267 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,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);

template<
    typename SyncWriteStream,
    typename ConstBufferSequence>
std::size_t write(
    SyncWriteStream & s,
    const ConstBufferSequence & buffers,
    asio::error_code & ec,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);

template<
    typename SyncWriteStream,
    typename ConstBufferSequence,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    const ConstBufferSequence & buffers,
```

```

CompletionCondition completion_condition,
typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);

template<
    typename SyncWriteStream,
    typename ConstBufferSequence,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    const ConstBufferSequence & buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);

template<
    typename SyncWriteStream,
    typename DynamicBuffer>
std::size_t write(
    SyncWriteStream & s,
    DynamicBuffer && buffers,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

template<
    typename SyncWriteStream,
    typename DynamicBuffer>
std::size_t write(
    SyncWriteStream & s,
    DynamicBuffer && buffers,
    asio::error_code & ec,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

template<
    typename SyncWriteStream,
    typename DynamicBuffer,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

template<
    typename SyncWriteStream,
    typename DynamicBuffer,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);

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

5.267.1 write (1 of 12 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,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);

```

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());
```

5.267.2 write (2 of 12 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,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);
```

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);
```

5.267.3 write (3 of 12 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,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);
```

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.

5.267.4 write (4 of 12 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,
    typename enable_if< is_const_buffer_sequence< ConstBufferSequence >::value >::type * = 0);
```

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.

5.267.5 write (5 of 12 overloads)

Write all of the supplied data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename DynamicBuffer>
std::size_t write(
    SyncWriteStream & s,
    DynamicBuffer && buffers,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 dynamic buffer sequence 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.

buffers The dynamic buffer sequence from which data will be written. Successfully written data is automatically consumed from the buffers.

Return Value

The number of bytes transferred.

Exceptions

asio::system_error Thrown on failure.

Remarks

This overload is equivalent to calling:

```
asio::write(
    s, buffers,
    asio::transfer_all());
```

5.267.6 write (6 of 12 overloads)

Write all of the supplied data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename DynamicBuffer>
std::size_t write(
    SyncWriteStream & s,
    DynamicBuffer && buffers,
    asio::error_code & ec,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 dynamic buffer sequence 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.

buffers The dynamic buffer sequence from which data will be written. Successfully written data is automatically consumed from the buffers.

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, buffers,
    asio::transfer_all(), ec);
```

5.267.7 write (7 of 12 overloads)

Write a certain amount of data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename DynamicBuffer,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 dynamic buffer sequence 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.

buffers The dynamic buffer sequence from which data will be written. Successfully written data is automatically consumed from the buffers.

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.

5.267.8 write (8 of 12 overloads)

Write a certain amount of data to a stream before returning.

```
template<
    typename SyncWriteStream,
    typename DynamicBuffer,
    typename CompletionCondition>
std::size_t write(
    SyncWriteStream & s,
    DynamicBuffer && buffers,
    CompletionCondition completion_condition,
    asio::error_code & ec,
    typename enable_if< is_dynamic_buffer< DynamicBuffer >::value >::type * = 0);
```

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 dynamic buffer sequence 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.

buffers The dynamic buffer sequence from which data will be written. Successfully written data is automatically consumed from the buffers.

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.

5.267.9 write (9 of 12 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());
```

5.267.10 write (10 of 12 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);
```

5.267.11 write (11 of 12 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.

5.267.12 write (12 of 12 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.

5.268 `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

5.268.1 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());
```

5.268.2 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);
```

5.268.3 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.

5.268.4 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.

5.268.5 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());
```

5.268.6 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);
```

5.268.7 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.

5.268.8 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.

5.269 `yield_context`

Context object that represents the currently executing coroutine.

```
typedef basic_yield_context< unspecified > yield_context;
```

Types

Name	Description
<code>callee_type</code>	The coroutine callee type, used by the implementation.
<code>caller_type</code>	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. Construct a yield context from another yield context type.
<code>operator[]</code>	Return a yield context that sets the specified <code>error_code</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 Networking TS compatibility

Asio now provides the interfaces and functionality specified by the "C++ Extensions for Networking" Technical Specification. In addition to access via the usual Asio header files, this functionality may be accessed through special headers that correspond to the header files defined in the TS. These are listed in the table below:

Networking TS header	Asio header
#include <buffer>	#include <asio/ts/buffer.hpp>
#include <executor>	#include <asio/ts/executor.hpp>
#include <internet>	#include <asio/ts/internet.hpp>
#include <io_context>	#include <asio/ts/io_context.hpp>
#include <net>	#include <asio/ts/net.hpp>
#include <netfwd>	#include <asio/ts/netfwd.hpp>
#include <socket>	#include <asio/ts/socket.hpp>
#include <timer>	#include <asio/ts/timer.hpp>

In some cases the new Networking TS compatible interfaces supersede older Asio facilities. In these cases the older interfaces have been deprecated. The table below shows the new Networking TS interfaces and the facilities they replace:

New interface	Old interface	Notes
io_context	io_service	The name <code>io_service</code> is retained as a typedef.
dispatch	io_service::dispatch	The <code>dispatch</code> free function can be used to submit functions to any <code>Executor</code> or <code>ExecutionContext</code> .
post	io_service::post	The <code>dispatch</code> free function can be used to submit functions to any <code>Executor</code> or <code>ExecutionContext</code> .
defer	io_service::post when the <code>asio_handler_is_continuation</code> hook returns true	The <code>defer</code> free function can be used to submit functions to any <code>Executor</code> or <code>ExecutionContext</code> .
io_context::poll	io_service::poll overload that takes <code>error_code&</code>	The <code>error_code</code> overload is not required.
io_context::poll_one	io_service::poll_one overload that takes <code>error_code&</code>	The <code>error_code</code> overload is not required.
io_context::run	io_service::run overload that takes <code>error_code&</code>	The <code>error_code</code> overload is not required.
io_context::run_one	io_service::run_one overload that takes <code>error_code&</code>	The <code>error_code</code> overload is not required.
io_context::run_for, io_context::run_until, io_context::run_one_for, and io_context::run_one_until		These functions add the ability to run an <code>io_context</code> for a limited time.
io_context::restart	io_service::reset	

New interface	Old interface	Notes
execution_context, execution_context::service, and execution_context::id	io_service, io_service::service, and io_service::id	The service-related functionality has been moved to the execution_context base class. This may also be used as a base for creating custom execution contexts.
make_service	add_service	
strand	io_service::strand	This template works with any valid executor, and is itself a valid executor.
executor_work_guard and make_work_guard	io_service::work	Work tracking is now covered by the Executor requirements. These templates work with any valid executor.
executor_binder and bind_executor	io_service::wrap and io_service::strand::wrap	These templates work with any valid executor.
async_result with CompletionToken and Signature template parameters	handler_type and single parameter async_result	The async_result trait is now the single point of customisation for asynchronous operation completion handlers and return type.
associated_executor and get_associated_executor	asio_handler_invoke	The handler invocation hook has been replaced by the new Executor requirements and the associated executor traits.
associated_allocator and get_associated_allocator	asio_handler_allocate and asio_handler_deallocate	The handler allocation hooks have been replaced by the standard Allocator requirements and the associated allocator traits.
const_buffer::data and mutable_buffer::data	buffer_cast	
const_buffer::size and mutable_buffer::size	buffer_size for single buffers	buffer_size is not deprecated for single buffers as const_buffer and mutable_buffer now satisfy the buffer sequence requirements
const_buffer	const_buffers_1	The ConstBufferSequence requirements have been modified such that const_buffer now satisfies them.
mutable_buffer	mutable_buffers_1	The MutableBufferSequence requirements have been modified such that mutable_buffer now satisfies them.
basic_socket::get_executor (and corresponding member for I/O objects such as timers, serial ports, etc.)	basic_io_object::get_io_service	Use get_executor().context() to obtain the associated io_context.
socket_base::max_listen_connections	socket_base::max_connections	

New interface	Old interface	Notes
<code>socket_base::wait_type</code> , <code>basic_socket::wait</code> , <code>basic_socket::async_wait</code> , <code>basic_socket_acceptor::wait</code> , and <code>basic_socket_acceptor::async_wait</code>	<code>null_buffers</code>	Operations for reactive I/O.
<code>basic_socket_acceptor::accept</code> returns a socket	<code>basic_socket_acceptor::accept</code> takes a socket by reference	Uses move support so requires C++11 or later. To accept a connection into a socket object on a different <code>io_context</code> , pass the destination context to <code>accept</code> .
<code>basic_socket_acceptor::async_accept</code> passes socket to handler	<code>basic_socket_acceptor::async_accept</code> takes a socket by reference	Uses move support so requires C++11 or later. To accept a connection into a socket object on a different <code>io_context</code> , pass the destination context to <code>async_accept</code> .
<code>connect</code> overloads that take a range	<code>connect</code> overloads that take a single iterator	The <code>ip::basic_resolver::resolve</code> function now returns a range. When the <code>resolve</code> function's result is passed directly to <code>connect</code> , the range overload will be selected.
<code>async_connect</code> overloads that take a range	<code>async_connect</code> overloads that take a single iterator	The <code>ip::basic_resolver::resolve</code> function now returns a range. When the <code>resolve</code> function's result is passed directly to <code>async_connect</code> , the range overload will be selected.
<code>basic_socket_streambuf::duration</code>	<code>basic_socket_streambuf::duration_type</code>	
<code>basic_socket_streambuf::time_point</code>	<code>basic_socket_streambuf::time_type</code>	
<code>basic_socket_streambuf::expiry</code>	<code>basic_socket_streambuf::expires_at</code> and <code>basic_socket_streambuf::expires_from_now</code> getters	
<code>basic_socket_streambuf::expires_after</code>	<code>basic_socket_streambuf::expires_from_now</code> setter	
<code>basic_socket_streambuf::error</code>	<code>basic_socket_streambuf::puberror</code>	
<code>basic_socket_iostream::duration</code>	<code>basic_socket_iostream::duration_type</code>	
<code>basic_socket_iostream::time_point</code>	<code>basic_socket_iostream::time_type</code>	
<code>basic_socket_iostream::expiry</code>	<code>basic_socket_iostream::expires_at</code> and <code>basic_socket_iostream::expires_from_now</code> getters	
<code>basic_socket_iostream::expires_after</code>	<code>basic_socket_iostream::expires_from_now</code> setter	
<code>basic_waitable_timer::cancel</code>	<code>basic_waitable_timer::cancel</code> overload that takes <code>error_code&</code>	The <code>error_code</code> overload is not required.

New interface	Old interface	Notes
<code>basic_waitable_timer::cancel_one</code>	<code>basic_waitable_timer::cancel_one</code> overload that takes <code>error_code&</code>	The <code>error_code</code> overload is not required.
<code>basic_waitable_timer::expires_at</code> setter	<code>basic_waitable_timer::expires_at</code> setter that takes <code>error_code&</code>	The <code>error_code</code> overload is not required.
<code>basic_waitable_timer::expiry</code>	<code>basic_waitable_timer::expires_at</code> and <code>basic_waitable_timer::expires_from_now</code> getters	
<code>basic_waitable_timer::expires_after</code>	<code>basic_waitable_timer::expires_from_now</code> setter	
<code>ip::make_address</code>	<code>ip::address::from_string</code>	
<code>ip::make_address_v4</code>	<code>ip::address_v4::from_string</code> and <code>ip::address_v6::to_v4</code>	
<code>ip::make_address_v6</code>	<code>ip::address_v6::from_string</code> and <code>ip::address_v6::v4_mapped</code>	
<code>ip::address::to_string</code>	<code>ip::address::to_string</code> that takes <code>error_code&</code>	The <code>error_code</code> overload is not required.
<code>ip::address_v4::to_string</code>	<code>ip::address_v4::to_string</code> that takes <code>error_code&</code>	The <code>error_code</code> overload is not required.
<code>ip::address_v6::to_string</code>	<code>ip::address_v6::to_string</code> that takes <code>error_code&</code>	The <code>error_code</code> overload is not required.
No replacement	<code>ip::address_v6::is_v4_compatible</code> and <code>ip::address_v6::v4_compatible</code>	
<code>ip::network_v4</code>	<code>ip::address_v4::broadcast</code> , <code>ip::address_v4::is_class_a</code> , <code>ip::address_v4::is_class_b</code> , <code>ip::address_v4::is_class_c</code> , and <code>ip::address_v4::netmask</code>	The <code>network_v4</code> class adds the ability to manipulate IPv4 network addresses using CIDR notation.
<code>ip::network_v6</code>		The <code>network_v6</code> class adds the ability to manipulate IPv6 network addresses using CIDR notation.
<code>ip::address_v4_iterator</code> and <code>ip::address_v4_range</code>		The <code>ip::address_v4_iterator</code> and <code>address_v4_range</code> classes add the ability to iterate over all, or a subset of, IPv4 addresses.
<code>ip::address_v6_iterator</code> and <code>ip::address_v6_range</code>		The <code>ip::address_v6_iterator</code> and <code>address_v6_range</code> classes add the ability to iterate over all, or a subset of, IPv6 addresses.
<code>ip::basic_resolver::results_type</code>	<code>ip::basic_resolver::iterator</code>	Resolvers now produce ranges rather than single iterators.

New interface	Old interface	Notes
ip::basic_resolver::resolve overloads taking hostname and service as arguments	ip::basic_resolver::resolve overloads taking a ip::basic_resolver::query	
ip::basic_resolver::resolve returns a range	ip::basic_resolver::resolve returns a single iterator	
ip::basic_resolver::async_resolve overloads taking hostname and service as arguments	ip::basic_resolver::async_resolve overloads taking a ip::basic_resolver::query	
ip::basic_resolver::async_resolve calls the handler with a range	ip::basic_resolver::async_resolve calls the handler with a single iterator	

7 Revision History

Asio 1.12.0

- Completed the interface changes to reflect the Networking TS ([N4656](#)).
 - See the [list](#) of new interfaces and, where applicable, the corresponding old interfaces that have been superseded.
 - The service template parameters, and the corresponding classes, are disabled by default. For example, instead of `basic_socket<Protocol, SocketService>` we now have simply `basic_socket<Protocol>`. The old interface can be enabled by defining the `(BOOST_)ASIO_ENABLE_OLD_SERVICES` macro.
- Added support for customised handler tracking.
- Added reactor-related (i.e. descriptor readiness) events to handler tracking.
- Added special [concurrency hint](#) values that may be used to disable locking on a per `io_context` basis.
- Enabled perfect forwarding for the first `ssl::stream<>` constructor argument.
- Added ability to release ownership of the underlying native socket. (Requires Windows 8.1 or later when using the I/O completion port backend.)

Asio 1.11.0

- Implemented changes to substantially reflect the Networking Library Proposal ([N4370](#)).
 - New `Executor` type requirements and classes to support an executor framework, including the `execution_context` base class, the `executor_work` class for tracking outstanding work, and the `executor` polymorphic wrapper. Free functions `dispatch()`, `post()` and `defer()` have been added and are used to submit function objects to executors.
 - Completion handlers now have an associated executor and associated allocator. The free function `wrap()` is used to associate an executor with a handler or other object. The handler hooks for allocation, invocation and continuation have been deprecated.
 - A `system_executor` class has been added as a default executor.
 - The `io_service` class is now derived from `execution_context` and implements the executor type requirements in its nested `executor_type` class. The member functions `dispatch()`, `post()`, `defer()` and `wrap()` have been deprecated. The `io_service::work` class has been deprecated.
 - The `io_service` member function `reset()` has been renamed to `restart()`. The old name is retained for backward compatibility but has been deprecated.
 - The `make_service<>()` function is now used to add a new service to an execution context such as an `io_service`. The `add_service()` function has been deprecated.
 - A new `strand<>` template has been added to allow strand functionality to be used with generic executor types.
 - I/O objects (such as sockets and timers) now provide access to their associated `io_service` via a `context()` member function. The `get_io_service()` member function is deprecated.
 - All asynchronous operations and executor operations now support move-only handlers. However, the deprecated `io_service::io_service::dispatch()`, `io_service::strand::post()` and `io_service::strand::dispatch()` functions still require copyable handlers.
 - Waitable timer objects are now movable.
 - Waitable timers, socket iostreams and socket streambufs now provide an `expiry()` member function for obtaining the expiry time. The accessors `expires_at()` and `expires_after()` have been deprecated, though those names are retained for the mutating members.
 - The `std::packaged_task` class template is now supported as a completion handler. The initiating operation automatically returns the future associated with the task. The `package()` function has been added as a convenient factory for packaged tasks.
 - Sockets, socket acceptors and descriptors now provide `wait()` and `async_wait()` operations that may be used to wait for readiness. The `null_buffers` type has been deprecated.

- The proposed error code enum classes are simulated using namespaces. Existing asio error codes now have a correspondence with the standard error conditions.
 - Conversion between IP address types, and conversion from string to address, is now supported via the `address_cast<>()`, `make_address()`, `make_address_v4()` and `make_address_v6()` free functions. The `from_string()`, `to_v4()`, `to_v6()` and `v4_mapped()` member functions have been deprecated.
 - A default-constructed `ip::address` now represents an invalid address value that is neither IPv4 nor IPv6.
 - New `buffer()` overloads that generate mutable buffers for non-const `string` objects.
 - Support for dynamic buffer sequences that automatically grow and shrink to accomodate data as it is read or written. This is a generic facility similar to the existing `asio::streambuf` class. This support includes:
 - * New `dynamic_string_buffer` and `dynamic_vector_buffer` adapter classes that meet the `DynamicBufferSequence` type requirements.
 - * New `dynamic_buffer()` factory functions for creating a dynamic buffer adapter for a `vector` or `string`.
 - * New overloads for the `read()`, `async_read()`, `write()` and `async_write()`, `read_until()` and `async_read_until()` free functions that directly support dynamic buffer sequences.
 - Support for networks and address ranges. Thanks go to Oliver Kowalke for contributing to the design and providing the implementation on which this facility is based. The following new classes have been added:
 - * `address_iterator_v4` for iterating across IPv4 addresses
 - * `address_iterator_v6` for iterating across IPv6 addresses
 - * `address_range_v4` to represent a range of IPv4 addresses
 - * `address_range_v6` to represent a range of IPv6 addresses
 - * `network_v4` for manipulating IPv4 CIDR addresses, e.g. 1.2.3.0/24
 - * `network_v6` for manipulating IPv6 CIDR addresses, e.g. ffe0:/120
 - New convenience headers in `<asio/ts/* .hpp>` that correspond to the headers in the proposal.
- Added a new, executor-aware `thread_pool` class.
 - Changed `spawn()` to be executor-aware.
 - Added a new `spawn()` overload that takes only a function object.
 - Changed `spawn()` and `yield_context` to permit nested calls to the completion handler.
 - Removed previously deprecated functions.
 - Added options for disabling TLS v1.1 and v1.2.
 - Changed the SSL wrapper to call the password callback when loading an in-memory key.
 - Changed the tutorial to use `std::endl` to ensure output is flushed.
 - 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.
 - Enabled perfect forwarding for the first `ssl::stream<>` constructor argument.
 - Added standalone Asio support for Clang when used with libstdc++ and C++11.
 - 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.
 - Ensured errors generated by Windows' ConnectEx function are mapped to their portable equivalents.
 - Changed multicast test to treat certain `join_group` failures as non-fatal.

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_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> f = 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(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 `_snwprintf` 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_type`, 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_)ASIO_HASH_MAP_SIZE` 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 `HANDLE`s such as named pipes (requires `HANDLE`s that work with I/O completion ports).
- Added wrappers for Windows random-access `HANDLE`s such as files (requires `HANDLE`s 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

-
~bad_address_cast
 ip::bad_address_cast, 954
~basic_datagram_socket
 basic_datagram_socket, 345
~basic_deadline_timer
 basic_deadline_timer, 359
~basic_io_object
 basic_io_object, 365
~basic_raw_socket
 basic_raw_socket, 432
~basic_resolver
 ip::basic_resolver, 1012
~basic_seq_packet_socket
 basic_seq_packet_socket, 489
~basic_socket
 basic_socket, 538
~basic_socket_acceptor
 basic_socket_acceptor, 587
~basic_socket_streambuf
 basic_socket_streambuf, 602
~basic_stream_socket
 basic_stream_socket, 666
~basic_waitable_timer
 basic_waitable_timer, 693
~context
 ssl::context, 1300
~context_base
 ssl::context_base, 1304
~descriptor
 posix::descriptor, 1153
~descriptor_base
 posix::descriptor_base, 1154
~error_category
 error_category, 803
~execution_context
 execution_context, 814
~executor
 executor, 823
~executor_binder
 executor_binder, 831
~executor_work_guard
 executor_work_guard, 833
~io_context
 io_context, 897
~overlapped_handle
 windows::overlapped_handle, 1395
~overlapped_ptr
 windows::overlapped_ptr, 1398
~resolver_base
 ip::resolver_base, 1069
~resolver_query_base
 ip::resolver_query_base, 1071
~serial_port
 serial_port, 1240
~serial_port_base
 serial_port_base, 1241
~service
 execution_context::service, 815
 io_context::service, 902
~signal_set
 signal_set, 1257
~socket_base
 socket_base, 1268
~strand
 io_context::strand, 909
 strand, 1337
~stream
 ssl::stream, 1323
~stream_base
 ssl::stream_base, 1325
~system_context
 system_context, 1346
~system_error
 system_error, 1348
~thread
 thread, 1356
~thread_pool
 thread_pool, 1363
~work
 io_context::work, 911

A
accept
 basic_socket_acceptor, 541
acceptor
 ip::tcp, 1072
 local::stream_protocol, 1115
access_denied
 error::basic_errors, 797
add
 signal_set, 1249
 time_traits< boost::posix_time::ptime >, 1368
add_certificate_authority
 ssl::context, 1275
add_service, 216
 execution_context, 809
 io_context, 883
 system_context, 1341
 thread_pool, 1358
add_verify_path
 ssl::context, 1276
address
 ip::address, 916
 ip::basic_endpoint, 981
 ip::network_v4, 1057

ip::network_v6, 1063
 address_configured
 ip::basic_resolver, 989
 ip::basic_resolver_query, 1024
 ip::resolver_base, 1068
 ip::resolver_query_base, 1070
 address_family_not_supported
 error::basic_errors, 797
 address_in_use
 error::basic_errors, 797
 address_v4
 ip::address_v4, 926
 address_v6
 ip::address_v6, 941
 all_matching
 ip::basic_resolver, 989
 ip::basic_resolver_query, 1024
 ip::resolver_base, 1068
 ip::resolver_query_base, 1070
 allocator_type
 use_future_t, 1372
 already_connected
 error::basic_errors, 797
 already_open
 error::misc_errors, 800
 already_started
 error::basic_errors, 797
 any
 ip::address_v4, 927
 ip::address_v6, 942
 argument_type
 executor_binder, 825
 asio_handler_allocate, 217
 asio_handler_deallocate, 217
 asio_handler_invoke, 218
 asio_handler_is_continuation, 219
 asn1
 ssl::context, 1279
 ssl::context_base, 1302
 assign
 basic_datagram_socket, 282
 basic_raw_socket, 368
 basic_seq_packet_socket, 435
 basic_socket, 493
 basic_socket_acceptor, 549
 basic_stream_socket, 606
 error_code, 804
 posix::descriptor, 1138
 posix::stream_descriptor, 1156
 serial_port, 1223
 windows::object_handle, 1377
 windows::overlapped_handle, 1387
 windows::random_access_handle, 1399
 windows::stream_handle, 1413
 async_accept
 basic_socket_acceptor, 550
 async_completion
 async_completion, 223
 async_connect, 224
 basic_datagram_socket, 283
 basic_raw_socket, 369
 basic_seq_packet_socket, 436
 basic_socket, 493
 basic_stream_socket, 607
 async_fill
 buffered_read_stream, 722
 buffered_stream, 730
 async_flush
 buffered_stream, 730
 buffered_write_stream, 739
 async_handshake
 ssl::stream, 1308
 async_read, 233
 async_read_at, 241
 async_read_some
 basic_stream_socket, 607
 buffered_read_stream, 722
 buffered_stream, 730
 buffered_write_stream, 739
 posix::stream_descriptor, 1157
 serial_port, 1224
 ssl::stream, 1310
 windows::stream_handle, 1414
 async_read_some_at
 windows::random_access_handle, 1400
 async_read_until, 246
 async_receive
 basic_datagram_socket, 283
 basic_raw_socket, 370
 basic_seq_packet_socket, 437
 basic_stream_socket, 608
 async_receive_from
 basic_datagram_socket, 285
 basic_raw_socket, 372
 async_resolve
 ip::basic_resolver, 989
 async_result
 async_result, 262
 async_result< Handler >, 263
 async_result< std::packaged_task< Result(Args...) >, Signature >, 264
 async_send
 basic_datagram_socket, 287
 basic_raw_socket, 373
 basic_seq_packet_socket, 439
 basic_stream_socket, 610
 async_send_to
 basic_datagram_socket, 289
 basic_raw_socket, 375
 async_shutdown
 ssl::stream, 1310
 async_wait

basic_datagram_socket, 290
basic_deadline_timer, 348
basic_raw_socket, 377
basic_seq_packet_socket, 439
basic_socket, 494
basic_socket_acceptor, 555
basic_stream_socket, 612
basic_waitable_timer, 681
posix::descriptor, 1139
posix::stream_descriptor, 1158
signal_set, 1250
windows::object_handle, 1378
async_write, 265
async_write_at, 272
async_write_some

- basic_stream_socket**, 613
- buffered_read_stream**, 722
- buffered_stream**, 731
- buffered_write_stream**, 739
- posix::stream_descriptor**, 1158
- serial_port**, 1225
- ssl::stream**, 1311
- windows::stream_handle**, 1415

async_write_some_at

- windows::random_access_handle**, 1401

at_mark

- basic_datagram_socket**, 291
- basic_raw_socket**, 378
- basic_seq_packet_socket**, 440
- basic_socket**, 495
- basic_stream_socket**, 614

available

- basic_datagram_socket**, 292
- basic_raw_socket**, 379
- basic_seq_packet_socket**, 441
- basic_socket**, 496
- basic_stream_socket**, 614

B
bad_address_cast

- ip::bad_address_cast**, 954

bad_descriptor

- error::basic_errors**, 797

bad_executor

- bad_executor**, 278

basic_address_iterator

- ip::basic_address_iterator< address_v4 >**, 955
- ip::basic_address_iterator< address_v6 >**, 964

basic_address_range

- ip::basic_address_range< address_v4 >**, 972
- ip::basic_address_range< address_v6 >**, 976

basic_datagram_socket

- basic_datagram_socket**, 293

basic_deadline_timer

- basic_deadline_timer**, 349

basic_endpoint

- generic::basic_endpoint**, 835

- ip::basic_endpoint**, 981
- local::basic_endpoint**, 1102

basic_io_object

- basic_io_object**, 360

basic_raw_socket

- basic_raw_socket**, 380

basic_resolver

- ip::basic_resolver**, 994

basic_resolver_entry

- ip::basic_resolver_entry**, 1013

basic_resolver_iterator

- ip::basic_resolver_iterator**, 1017

basic_resolver_query

- ip::basic_resolver_query**, 1024

basic_resolver_results

- ip::basic_resolver_results**, 1031

basic_seq_packet_socket

- basic_seq_packet_socket**, 442

basic_socket

- basic_socket**, 497

basic_socket_acceptor

- basic_socket_acceptor**, 556

basic_socket_iostream

- basic_socket_iostream**, 589

basic_socket_streambuf

- basic_socket_streambuf**, 596

basic_stream_socket

- basic_stream_socket**, 615

basic_streampbuf

- basic_streampbuf**, 668

basic_streampbuf_ref

- basic_streampbuf_ref**, 672

basic_waitable_timer

- basic_waitable_timer**, 681

basic_yield_context

- basic_yield_context**, 694

baud_rate

- serial_port_base::baud_rate**, 1242

begin

- buffers_iterator**, 747
- const_buffers_1**, 770
- ip::basic_address_range< address_v4 >**, 973
- ip::basic_address_range< address_v6 >**, 977
- ip::basic_resolver_results**, 1031
- mutable_buffers_1**, 1130
- null_buffers**, 1133

bind

- basic_datagram_socket**, 296
- basic_raw_socket**, 383
- basic_seq_packet_socket**, 445
- basic_socket**, 500
- basic_socket_acceptor**, 559
- basic_stream_socket**, 618

bind_executor, 696

broadcast

- basic_datagram_socket**, 297

basic_raw_socket, 384
basic_seq_packet_socket, 446
basic_socket, 501
basic_socket_acceptor, 561
basic_stream_socket, 620
ip::address_v4, 927
ip::network_v4, 1057
socket_base, 1259
broken_pipe
 error::basic_errors, 797
buffer, 697
buffer_cast, 714
buffer_copy, 715
buffer_sequence_begin, 717
buffer_sequence_end, 718
buffer_size, 719
buffered_read_stream
 buffered_read_stream, 722
buffered_stream
 buffered_stream, 731
buffered_write_stream
 buffered_write_stream, 740
buffers_begin, 745
buffers_end, 746
buffers_iterator
 buffers_iterator, 747
bytes_readable
 basic_datagram_socket, 298
 basic_raw_socket, 384
 basic_seq_packet_socket, 447
 basic_socket, 501
 basic_socket_acceptor, 561
 basic_stream_socket, 620
 posix::descriptor, 1139
 posix::descriptor_base, 1154
 posix::stream_descriptor, 1159
 socket_base, 1260
bytes_type
 ip::address_v4, 928
 ip::address_v6, 942

C

callee_type
 basic_yield_context, 695
caller_type
 basic_yield_context, 696
cancel
 basic_datagram_socket, 298
 basic_deadline_timer, 350
 basic_raw_socket, 385
 basic_seq_packet_socket, 447
 basic_socket, 502
 basic_socket_acceptor, 562
 basic_stream_socket, 621
 basic_waitable_timer, 683
 ip::basic_resolver, 995
 posix::descriptor, 1140

 posix::stream_descriptor, 1160
 serial_port, 1225
 signal_set, 1250
 windows::object_handle, 1378
 windows::overlapped_handle, 1388
 windows::random_access_handle, 1401
 windows::stream_handle, 1416

cancel_one
 basic_deadline_timer, 352
 basic_waitable_timer, 684

canonical
 ip::network_v4, 1057
 ip::network_v6, 1063

canonical_name
 ip::basic_resolver, 995
 ip::basic_resolver_query, 1027
 ip::resolver_base, 1068
 ip::resolver_query_base, 1070

capacity
 basic_streambuf, 669
 basic_streambuf_ref, 673
 dynamic_string_buffer, 787
 dynamic_vector_buffer, 792
 generic::basic_endpoint, 836
 ip::basic_endpoint, 982
 local::basic_endpoint, 1103

category
 error_code, 804

cbegin
 ip::basic_resolver_results, 1032

cend
 ip::basic_resolver_results, 1032

character_size
 serial_port_base::character_size, 1243

clear
 error_code, 805
 signal_set, 1252

clear_options
 ssl::context, 1277

client
 ssl::stream, 1314
 ssl::stream_base, 1325

clock_type
 basic_socket_iostream, 590
 basic_socket_streambuf, 597
 basic_waitable_timer, 686

close
 basic_datagram_socket, 300
 basic_raw_socket, 386
 basic_seq_packet_socket, 449
 basic_socket, 503
 basic_socket_acceptor, 562
 basic_socket_iostream, 590
 basic_socket_streambuf, 597
 basic_stream_socket, 622
 buffered_read_stream, 723

buffered_stream, 731
 buffered_write_stream, 740
 posix::descriptor, 1141
 posix::stream_descriptor, 1160
 serial_port, 1226
 windows::object_handle, 1379
 windows::overlapped_handle, 1389
 windows::random_access_handle, 1402
 windows::stream_handle, 1416
 code
 system_error, 1347
 commit
 basic_streambuf, 669
 basic_streambuf_ref, 673
 dynamic_string_buffer, 787
 dynamic_vector_buffer, 792
 complete
 windows::overlapped_ptr, 1396
 completion_handler
 async_completion, 223
 completion_handler_type
 async_completion, 223
 async_result, 262
 async_result< std::packaged_task< Result(Args...) >, Signature >, 265
 connect, 754
 basic_datagram_socket, 301
 basic_raw_socket, 387
 basic_seq_packet_socket, 450
 basic_socket, 504
 basic_socket_iostream, 590
 basic_socket_streambuf, 597
 basic_stream_socket, 623
 connection_aborted
 error::basic_errors, 797
 connection_refused
 error::basic_errors, 797
 connection_reset
 error::basic_errors, 797
 const_buffer
 const_buffer, 768
 const_buffers_1
 const_buffers_1, 770
 const_buffers_type
 basic_streambuf, 669
 basic_streambuf_ref, 673
 dynamic_string_buffer, 787
 dynamic_vector_buffer, 793
 const_iterator
 const_buffers_1, 771
 ip::basic_resolver_results, 1032
 mutable_buffers_1, 1130
 null_buffers, 1133
 const_reference
 ip::basic_resolver_results, 1033
 consume

basic_streambuf, 669
 basic_streambuf_ref, 675
 dynamic_string_buffer, 788
 dynamic_vector_buffer, 793
 context
 execution_context::service, 815
 executor, 817
 io_context::executor_type, 898
 io_context::strand, 904
 ssl::context, 1278
 strand, 1332
 system_executor, 1349
 thread_pool::executor_type, 1364
 coroutine
 coroutine, 777
 count_type
 io_context, 883

D

data
 basic_streambuf, 670
 basic_streambuf_ref, 675
 const_buffer, 769
 const_buffers_1, 771
 dynamic_string_buffer, 788
 dynamic_vector_buffer, 794
 generic::basic_endpoint, 836
 ip::basic_endpoint, 983
 local::basic_endpoint, 1103
 mutable_buffer, 1127
 mutable_buffers_1, 1130
 data_type
 generic::basic_endpoint, 837
 ip::basic_endpoint, 983
 local::basic_endpoint, 1103
 datagram_protocol
 generic::datagram_protocol, 840
 deadline_timer, 778
 debug
 basic_datagram_socket, 302
 basic_raw_socket, 389
 basic_seq_packet_socket, 451
 basic_socket, 506
 basic_socket_acceptor, 563
 basic_stream_socket, 624
 socket_base, 1260
 default_buffer_size
 buffered_read_stream, 723
 buffered_write_stream, 741
 default_workarounds
 ssl::context, 1279
 ssl::context_base, 1301
 defer, 780
 executor, 817
 io_context::executor_type, 898
 io_context::strand, 904
 strand, 1332

```

system_executor, 1349
thread_pool::executor_type, 1364
dereference
    ip::basic_resolver_iterator, 1018
    ip::basic_resolver_results, 1034
descriptor
    posix::descriptor, 1141
destroy
    execution_context, 810
    io_context, 884
    system_context, 1341
    thread_pool, 1358
difference_type
    buffers_iterator, 748
    ip::basic_address_iterator< address_v4 >, 956
    ip::basic_address_iterator< address_v6 >, 964
    ip::basic_resolver_iterator, 1018
    ip::basic_resolver_results, 1034
dispatch, 782
    executor, 818
    io_context, 884
    io_context::executor_type, 899
    io_context::strand, 904
    strand, 1332
    system_executor, 1350
    thread_pool::executor_type, 1365
do_not_route
    basic_datagram_socket, 303
    basic_raw_socket, 389
    basic_seq_packet_socket, 452
    basic_socket, 506
    basic_socket_acceptor, 564
    basic_stream_socket, 625
    socket_base, 1261
duration
    basic_socket_iostream, 590
    basic_socket_streambuf, 598
    basic_waitable_timer, 686
duration_type
    basic_deadline_timer, 353
    basic_socket_iostream, 591
    basic_socket_streambuf, 598
    time_traits< boost::posix_time::ptime >, 1368
dynamic_buffer, 784
dynamic_string_buffer
    dynamic_string_buffer, 789
dynamic_vector_buffer
    dynamic_vector_buffer, 794

E
empty
    ip::basic_address_range< address_v4 >, 973
    ip::basic_address_range< address_v6 >, 977
    ip::basic_resolver_results, 1035
enable_connection_aborted
    basic_datagram_socket, 303
    basic_raw_socket, 390
    basic_seq_packet_socket, 452
    basic_socket, 507
    basic_socket_acceptor, 564
    basic_stream_socket, 626
    socket_base, 1261
end
    buffers_iterator, 748
    const_buffers_1, 771
    ip::basic_address_range< address_v4 >, 973
    ip::basic_address_range< address_v6 >, 977
    ip::basic_resolver_results, 1035
    mutable_buffers_1, 1130
    null_buffers, 1134
endpoint
    generic::datagram_protocol, 841
    generic::raw_protocol, 848
    generic::seq_packet_protocol, 856
    generic::stream_protocol, 863
    ip::basic_resolver_entry, 1014
    ip::icmp, 1045
    ip::tcp, 1076
    ip::udp, 1087
    local::datagram_protocol, 1108
    local::stream_protocol, 1118
endpoint_type
    basic_datagram_socket, 304
    basic_raw_socket, 391
    basic_seq_packet_socket, 453
    basic_socket, 507
    basic_socket_acceptor, 565
    basic_socket_iostream, 591
    basic_socket_streambuf, 599
    basic_stream_socket, 626
    ip::basic_resolver, 995
    ip::basic_resolver_entry, 1014
    ip::basic_resolver_results, 1035
eof
    error::misc_errors, 800
equal
    ip::basic_resolver_iterator, 1018
    ip::basic_resolver_results, 1035
error
    basic_socket_iostream, 591
    basic_socket_streambuf, 599
error::addrinfo_category, 796
error::addrinfo_errors, 797
error::basic_errors, 797
error::get_addrinfo_category, 798
error::get_misc_category, 798
error::get_netdb_category, 799
error::get_ssl_category, 799
error::get_system_category, 799
error::make_error_code, 799
error::misc_category, 800
error::misc_errors, 800
error::netdb_category, 801

```

error::netdb_errors, 801
 error::ssl_category, 801
 error::ssl_errors, 802
 error::system_category, 802
 error_code
 error_code, 805
 even
 serial_port_base::parity, 1245
 execution_context
 execution_context, 810
 executor
 executor, 818
 executor_arg, 823
 executor_arg_t
 executor_arg_t, 824
 executor_binder
 executor_binder, 826
 executor_type
 basic_datagram_socket, 304
 basic_deadline_timer, 353
 basic_io_object, 362
 basic_raw_socket, 391
 basic_seq_packet_socket, 453
 basic_socket, 507
 basic_socket_acceptor, 565
 basic_stream_socket, 626
 basic_waitable_timer, 686
 buffered_read_stream, 723
 buffered_stream, 732
 buffered_write_stream, 741
 executor_binder, 829
 executor_work_guard, 832
 ip::basic_resolver, 995
 posix::descriptor, 1143
 posix::stream_descriptor, 1161
 serial_port, 1227
 signal_set, 1252
 ssl::stream, 1311
 system_context, 1341
 windows::object_handle, 1380
 windows::overlapped_handle, 1389
 windows::random_access_handle, 1403
 windows::stream_handle, 1417
 executor_work_guard
 executor_work_guard, 832
 expires_after
 basic_socket_iostream, 592
 basic_socket_streambuf, 599
 basic_waitable_timer, 687
 expires_at
 basic_deadline_timer, 354
 basic_socket_iostream, 592
 basic_socket_streambuf, 599
 basic_waitable_timer, 688
 expires_from_now
 basic_deadline_timer, 356
 basic_socket_iostream, 593
 basic_socket_streambuf, 600
 basic_waitable_timer, 690
 expiry
 basic_socket_iostream, 593
 basic_socket_streambuf, 601
 basic_waitable_timer, 691

F

family
 generic::datagram_protocol, 842
 generic::raw_protocol, 850
 generic::seq_packet_protocol, 857
 generic::stream_protocol, 865
 ip::icmp, 1047
 ip::tcp, 1077
 ip::udp, 1089
 local::datagram_protocol, 1110
 local::stream_protocol, 1119

fault
 error::basic_errors, 797

fd_set_failure
 error::misc_errors, 800

file_format
 ssl::context, 1279
 ssl::context_base, 1302

fill
 buffered_read_stream, 724
 buffered_stream, 732

find
 ip::basic_address_range< address_v4 >, 974
 ip::basic_address_range< address_v6 >, 977

first_argument_type
 executor_binder, 829

flags
 ip::basic_resolver, 996
 ip::basic_resolver_query, 1027
 ip::resolver_base, 1068
 ip::resolver_query_base, 1070

flow_control
 serial_port_base::flow_control, 1244

flush
 buffered_stream, 733
 buffered_write_stream, 741

for_reading
 ssl::context, 1283
 ssl::context_base, 1304

for_writing
 ssl::context, 1283
 ssl::context_base, 1304

fork_child
 execution_context, 810
 io_context, 884
 system_context, 1342
 thread_pool, 1358

fork_event
 execution_context, 810

io_context, 884
 system_context, 1342
 thread_pool, 1358
fork_parent
 execution_context, 810
 io_context, 884
 system_context, 1342
 thread_pool, 1358
fork_prepare
 execution_context, 810
 io_context, 884
 system_context, 1342
 thread_pool, 1358
from_string
 ip::address, 917
 ip::address_v4, 928
 ip::address_v6, 942

G

get
 associated_allocator, 220
 associated_executor, 221
 async_result, 262
 async_result< Handler >, 263
 async_result< std::packaged_task< Result(Args...) >, Signature >, 265
 executor_binder, 829
 windows::overlapped_ptr, 1396

get_allocator
 use_future_t, 1372

get_associated_allocator, 871

get_associated_executor, 872

get_executor
 basic_datagram_socket, 305
 basic_deadline_timer, 357
 basic_io_object, 362
 basic_raw_socket, 392
 basic_seq_packet_socket, 454
 basic_socket, 508
 basic_socket_acceptor, 566
 basic_stream_socket, 627
 basic_waitable_timer, 691
 buffered_read_stream, 724
 buffered_stream, 733
 buffered_write_stream, 741
 executor_binder, 830
 executor_work_guard, 833
 io_context, 885
 ip::basic_resolver, 996
 posix::descriptor, 1143
 posix::stream_descriptor, 1162
 serial_port, 1228
 signal_set, 1253
 ssl::stream, 1311
 system_context, 1343
 thread_pool, 1359
 windows::object_handle, 1380

windows::overlapped_handle, 1390
 windows::random_access_handle, 1404
 windows::stream_handle, 1418

get_implementation
 basic_io_object, 362

get_inner_executor
 strand, 1333

get_io_context
 basic_datagram_socket, 305
 basic_deadline_timer, 357
 basic_io_object, 363
 basic_raw_socket, 392
 basic_seq_packet_socket, 454
 basic_socket, 508
 basic_socket_acceptor, 566
 basic_stream_socket, 627
 basic_waitable_timer, 691
 buffered_read_stream, 724
 buffered_stream, 733
 buffered_write_stream, 742
 io_context::service, 901
 io_context::strand, 906
 io_context::work, 910
 ip::basic_resolver, 997
 posix::descriptor, 1143
 posix::stream_descriptor, 1162
 serial_port, 1228
 signal_set, 1253
 ssl::stream, 1312
 windows::object_handle, 1380
 windows::overlapped_handle, 1390
 windows::random_access_handle, 1404
 windows::stream_handle, 1418

get_io_service
 basic_datagram_socket, 305
 basic_deadline_timer, 358
 basic_io_object, 363
 basic_raw_socket, 392
 basic_seq_packet_socket, 454
 basic_socket, 509
 basic_socket_acceptor, 566
 basic_stream_socket, 628
 basic_waitable_timer, 692
 buffered_read_stream, 724
 buffered_stream, 733
 buffered_write_stream, 742
 io_context::service, 901
 io_context::strand, 906
 io_context::work, 910
 ip::basic_resolver, 997
 posix::descriptor, 1144
 posix::stream_descriptor, 1162
 serial_port, 1228
 signal_set, 1254
 ssl::stream, 1312
 windows::object_handle, 1381

windows::overlapped_handle, 1390
 windows::random_access_handle, 1404
 windows::stream_handle, 1418

get_option
 basic_datagram_socket, 306
 basic_raw_socket, 392
 basic_seq_packet_socket, 455
 basic_socket, 509
 basic_socket_acceptor, 567
 basic_stream_socket, 628
 serial_port, 1228

get_service
 basic_io_object, 363

H

handshake
 ssl::stream, 1312

handshake_type
 ssl::stream, 1314
 ssl::stream_base, 1325

hardware
 serial_port_base::flow_control, 1244

has_service, 875
 execution_context, 810
 io_context, 885
 system_context, 1343
 thread_pool, 1359

high_resolution_timer, 875

hints
 ip::basic_resolver_query, 1027

host_name
 ip::basic_resolver_entry, 1014
 ip::basic_resolver_query, 1028

host_not_found
 error::netdb_errors, 801

host_not_found_try_again
 error::netdb_errors, 801

host_unreachable
 error::basic_errors, 797

hosts
 ip::network_v4, 1057
 ip::network_v6, 1063

I

id
 execution_context::id, 814

implementation_type
 basic_io_object, 364

in_avail
 buffered_read_stream, 725
 buffered_stream, 734
 buffered_write_stream, 742

in_progress
 error::basic_errors, 797

increment
 ip::basic_resolver_iterator, 1018
 ip::basic_resolver_results, 1035

index_
 ip::basic_resolver_iterator, 1018
 ip::basic_resolver_results, 1035

inner_executor_type
 strand, 1333

interrupted
 error::basic_errors, 797

invalid_argument
 error::basic_errors, 797

invalid_service_owner
 invalid_service_owner, 878

io_context
 io_context, 885

io_control
 basic_datagram_socket, 307
 basic_raw_socket, 394
 basic_seq_packet_socket, 456
 basic_socket, 510
 basic_socket_acceptor, 568
 basic_stream_socket, 629
 posix::descriptor, 1144
 posix::stream_descriptor, 1163

io_service, 911

iostream
 generic::stream_protocol, 865
 ip::tcp, 1078
 local::stream_protocol, 1119

ip::address_v4_iterator, 936
 ip::address_v4_range, 938
 ip::address_v6_iterator, 951
 ip::address_v6_range, 952
 ip::host_name, 1043
 ip::multicast::enable_loopback, 1053
 ip::multicast::hops, 1054
 ip::multicast::join_group, 1054
 ip::multicast::leave_group, 1055
 ip::multicast::outbound_interface, 1055
 ip::unicast::hops, 1095
 ip::v4_mapped_t, 1096
 ip::v6_only, 1096

is_child
 coroutine, 777

is_class_a
 ip::address_v4, 929

is_class_b
 ip::address_v4, 929

is_class_c
 ip::address_v4, 929

is_complete
 coroutine, 777

is_host
 ip::network_v4, 1057
 ip::network_v6, 1063

is_link_local
 ip::address_v6, 943

is_loopback

ip::address, 918
 ip::address_v4, 929
 ip::address_v6, 943
 is_multicast
 ip::address, 919
 ip::address_v4, 929
 ip::address_v6, 943
 is_multicast_global
 ip::address_v6, 943
 is_multicast_link_local
 ip::address_v6, 943
 is_multicast_node_local
 ip::address_v6, 943
 is_multicast_org_local
 ip::address_v6, 944
 is_multicast_site_local
 ip::address_v6, 944
 is_open
 basic_datagram_socket, 309
 basic_raw_socket, 395
 basic_seq_packet_socket, 458
 basic_socket, 511
 basic_socket_acceptor, 569
 basic_stream_socket, 631
 posix::descriptor, 1145
 posix::stream_descriptor, 1164
 serial_port, 1229
 windows::object_handle, 1381
 windows::overlapped_handle, 1391
 windows::random_access_handle, 1404
 windows::stream_handle, 1419
 is_parent
 coroutine, 778
 is_site_local
 ip::address_v6, 944
 is_subnet_of
 ip::network_v4, 1058
 ip::network_v6, 1063
 is_unspecified
 ip::address, 919
 ip::address_v4, 930
 ip::address_v6, 944
 is_v4
 ip::address, 919
 is_v4_compatible
 ip::address_v6, 944
 is_v4_mapped
 ip::address_v6, 944
 is_v6
 ip::address, 919
 iterator
 ip::basic_address_range< address_v4 >, 974
 ip::basic_address_range< address_v6 >, 977
 ip::basic_resolver, 997
 ip::basic_resolver_results, 1035
 iterator_category
 buffers_iterator, 748
 ip::basic_address_iterator< address_v4 >, 956
 ip::basic_address_iterator< address_v6 >, 965
 ip::basic_resolver_iterator, 1018
 ip::basic_resolver_results, 1037

J

join
 system_context, 1343
 thread, 1355
 thread_pool, 1359

K

keep_alive
 basic_datagram_socket, 309
 basic_raw_socket, 396
 basic_seq_packet_socket, 458
 basic_socket, 512
 basic_socket_acceptor, 570
 basic_stream_socket, 631
 socket_base, 1262

L

less_than
 time_traits< boost::posix_time::ptime >, 1368

linger
 basic_datagram_socket, 310
 basic_raw_socket, 396
 basic_seq_packet_socket, 459
 basic_socket, 512
 basic_socket_acceptor, 570
 basic_stream_socket, 632
 socket_base, 1262

listen
 basic_socket_acceptor, 571

load
 serial_port_base::baud_rate, 1242
 serial_port_base::character_size, 1243
 serial_port_base::flow_control, 1244
 serial_port_base::parity, 1245
 serial_port_base::stop_bits, 1246

load_verify_file
 ssl::context, 1279

local::connect_pair, 1107

local_endpoint
 basic_datagram_socket, 310
 basic_raw_socket, 397
 basic_seq_packet_socket, 459
 basic_socket, 513
 basic_socket_acceptor, 572
 basic_stream_socket, 632

loopback
 ip::address_v4, 930
 ip::address_v6, 944

lowest_layer
 basic_datagram_socket, 311
 basic_raw_socket, 398

basic_seq_packet_socket, 460
 basic_socket, 514
 basic_stream_socket, 633
 buffered_read_stream, 725
 buffered_stream, 734
 buffered_write_stream, 742
 posix::descriptor, 1146
 posix::stream_descriptor, 1164
 serial_port, 1229
 ssl::stream, 1314
 windows::object_handle, 1381
 windows::overlapped_handle, 1391
 windows::random_access_handle, 1405
 windows::stream_handle, 1419
lowest_layer_type
 basic_datagram_socket, 312
 basic_raw_socket, 399
 basic_seq_packet_socket, 461
 basic_socket, 515
 basic_stream_socket, 634
 buffered_read_stream, 725
 buffered_stream, 734
 buffered_write_stream, 743
 posix::descriptor, 1146
 posix::stream_descriptor, 1165
 serial_port, 1230
 ssl::stream, 1315
 windows::object_handle, 1382
 windows::overlapped_handle, 1391
 windows::random_access_handle, 1405
 windows::stream_handle, 1419

M

make_address
 ip::address, 919
 make_address_v4
 ip::address_v4, 930
 make_address_v6
 ip::address_v6, 945
 make_network_v4
 ip::address_v4, 932
 ip::network_v4, 1058
 make_network_v6
 ip::address_v6, 946
 ip::network_v6, 1063
 make_service
 execution_context, 811
 io_context, 886
 system_context, 1343
 thread_pool, 1360
 make_work_guard, 1124
 max_connections
 basic_datagram_socket, 316
 basic_raw_socket, 402
 basic_seq_packet_socket, 465
 basic_socket, 518
 basic_socket_acceptor, 573
 basic_stream_socket, 638
 socket_base, 1263
 max_listen_connections
 basic_datagram_socket, 316
 basic_raw_socket, 402
 basic_seq_packet_socket, 465
 basic_socket, 518
 basic_socket_acceptor, 573
 basic_stream_socket, 638
 socket_base, 1263
 max_size
 basic_streampbuf, 670
 basic_streampbuf_ref, 675
 dynamic_string_buffer, 789
 dynamic_vector_buffer, 795
 ip::basic_resolver_results, 1037
 message
 error_category, 803
 error_code, 805
 message_do_not_route
 basic_datagram_socket, 316
 basic_raw_socket, 402
 basic_seq_packet_socket, 465
 basic_socket, 518
 basic_socket_acceptor, 573
 basic_stream_socket, 638
 socket_base, 1263
 message_end_of_record
 basic_datagram_socket, 316
 basic_raw_socket, 402
 basic_seq_packet_socket, 465
 basic_socket, 518
 basic_socket_acceptor, 573
 basic_stream_socket, 638
 socket_base, 1263
 message_flags
 basic_datagram_socket, 316
 basic_raw_socket, 403
 basic_seq_packet_socket, 465
 basic_socket, 518
 basic_socket_acceptor, 574
 basic_stream_socket, 638
 socket_base, 1263
 message_out_of_band
 basic_datagram_socket, 316
 basic_raw_socket, 403
 basic_seq_packet_socket, 465
 basic_socket, 519
 basic_socket_acceptor, 574
 basic_stream_socket, 638
 socket_base, 1264
 message_peek
 basic_datagram_socket, 317
 basic_raw_socket, 403
 basic_seq_packet_socket, 466
 basic_socket, 519

basic_socket_acceptor, 574
basic_stream_socket, 639
socket_base, 1264
message_size
error::basic_errors, 797
method
ssl::context, 1280
ssl::context_base, 1302
mutable_buffer
mutable_buffer, 1127
mutable_buffers_1
mutable_buffers_1, 1130
mutable_buffers_type
basic_streampbuf, 670
basic_streampbuf_ref, 676
dynamic_string_buffer, 790
dynamic_vector_buffer, 795

N

name
error_category, 803
name_too_long
error::basic_errors, 797
native_handle
basic_datagram_socket, 317
basic_raw_socket, 403
basic_seq_packet_socket, 466
basic_socket, 519
basic_socket_acceptor, 574
basic_stream_socket, 639
posix::descriptor, 1148
posix::stream_descriptor, 1167
serial_port, 1232
ssl::context, 1281
ssl::stream, 1315
ssl::verify_context, 1326
windows::object_handle, 1383
windows::overlapped_handle, 1393
windows::random_access_handle, 1407
windows::stream_handle, 1421

native_handle_type
basic_datagram_socket, 317
basic_raw_socket, 403
basic_seq_packet_socket, 466
basic_socket, 519
basic_socket_acceptor, 574
basic_stream_socket, 639
posix::descriptor, 1148
posix::stream_descriptor, 1167
serial_port, 1232
ssl::context, 1281
ssl::stream, 1315
ssl::verify_context, 1326
windows::object_handle, 1383
windows::overlapped_handle, 1393
windows::random_access_handle, 1407
windows::stream_handle, 1421

native_non_blocking
basic_datagram_socket, 317
basic_raw_socket, 404
basic_seq_packet_socket, 466
basic_socket, 519
basic_socket_acceptor, 575
basic_stream_socket, 639
posix::descriptor, 1148
posix::stream_descriptor, 1167

netmask
ip::address_v4, 932
ip::network_v4, 1059

network
ip::network_v4, 1059
ip::network_v6, 1065

network_down
error::basic_errors, 797

network_reset
error::basic_errors, 797

network_unreachable
error::basic_errors, 797

network_v4
ip::network_v4, 1059

network_v6
ip::network_v6, 1065

next_layer
buffered_read_stream, 726
buffered_stream, 735
buffered_write_stream, 743
ssl::stream, 1315

next_layer_type
buffered_read_stream, 726
buffered_stream, 735
buffered_write_stream, 743
ssl::stream, 1316

no_buffer_space
error::basic_errors, 797

no_compression
ssl::context, 1281
ssl::context_base, 1303

no_data
error::netdb_errors, 801

no_delay
ip::tcp, 1079

no_descriptors
error::basic_errors, 797

no_memory
error::basic_errors, 797

no_permission
error::basic_errors, 797

no_protocol_option
error::basic_errors, 797

no_recovery
error::netdb_errors, 801

no_sslv2
ssl::context, 1281

ssl::context_base, 1303
 no_sslv3
 ssl::context, 1282
 ssl::context_base, 1303
 no_such_device
 error::basic_errors, 797
 no_tls1
 ssl::context, 1282
 ssl::context_base, 1303
 no_tls1_1
 ssl::context, 1282
 ssl::context_base, 1303
 no_tls1_2
 ssl::context, 1282
 ssl::context_base, 1303
 non_blocking
 basic_datagram_socket, 322
 basic_raw_socket, 409
 basic_seq_packet_socket, 471
 basic_socket, 524
 basic_socket_acceptor, 576
 basic_stream_socket, 644
 posix::descriptor, 1150
 posix::stream_descriptor, 1169
 none
 serial_port_base::flow_control, 1244
 serial_port_base::parity, 1245
 not_connected
 error::basic_errors, 797
 not_found
 error::misc_errors, 800
 not_socket
 error::basic_errors, 797
 notify_fork
 execution_context, 811
 execution_context::service, 816
 io_context, 887
 system_context, 1344
 thread_pool, 1360
 now
 time_traits< boost::posix_time::ptime >, 1368
 numeric_host
 ip::basic_resolver, 999
 ip::basic_resolver_query, 1028
 ip::resolver_base, 1068
 ip::resolver_query_base, 1071
 numeric_service
 ip::basic_resolver, 999
 ip::basic_resolver_query, 1028
 ip::resolver_base, 1068
 ip::resolver_query_base, 1071

O

object_handle
 windows::object_handle, 1384
 odd
 serial_port_base::parity, 1245

on_work_finished
 executor, 820
 io_context::executor_type, 899
 io_context::strand, 906
 strand, 1333
 system_executor, 1350
 thread_pool::executor_type, 1365

on_work_started
 executor, 820
 io_context::executor_type, 899
 io_context::strand, 906
 strand, 1333
 system_executor, 1350
 thread_pool::executor_type, 1366

one
 serial_port_base::stop_bits, 1246

onepointfive
 serial_port_base::stop_bits, 1246

open
 basic_datagram_socket, 324
 basic_raw_socket, 410
 basic_seq_packet_socket, 473
 basic_socket, 526
 basic_socket_acceptor, 577
 basic_stream_socket, 646
 serial_port, 1232

operation_aborted
 error::basic_errors, 797

operation_not_supported
 error::basic_errors, 797

operator *
 buffers_iterator, 748
 ip::basic_address_iterator< address_v4 >, 956
 ip::basic_address_iterator< address_v6 >, 965
 ip::basic_resolver_iterator, 1019
 ip::basic_resolver_results, 1037

operator endpoint_type
 ip::basic_resolver_entry, 1015

operator unspecified_bool_type
 error_code, 806
 executor, 820

operator!
 error_code, 806

operator!=
 buffers_iterator, 748
 error_category, 803
 error_code, 806
 executor, 820
 generic::basic_endpoint, 837
 generic::datagram_protocol, 843
 generic::raw_protocol, 850
 generic::seq_packet_protocol, 857
 generic::stream_protocol, 866
 io_context::executor_type, 900
 io_context::strand, 906
 ip::address, 920

ip::address_v4, 932
 ip::address_v6, 947
 ip::basic_address_iterator< address_v4 >, 956
 ip::basic_address_iterator< address_v6 >, 965
 ip::basic_endpoint, 983
 ip::basic_resolver_iterator, 1019
 ip::basic_resolver_results, 1038
 ip::icmp, 1047
 ip::network_v4, 1060
 ip::network_v6, 1065
 ip::tcp, 1079
 ip::udp, 1089
 local::basic_endpoint, 1103
 strand, 1333
 system_executor, 1350
 thread_pool::executor_type, 1366
operator()
 executor_binder, 830
 ssl::rfc2818_verification, 1306
 use_future_t, 1372
operator+
 buffers_iterator, 749
 const_buffer, 769
 const_buffers_1, 772
 mutable_buffer, 1128
 mutable_buffers_1, 1131
operator++
 buffers_iterator, 749
 ip::basic_address_iterator< address_v4 >, 957
 ip::basic_address_iterator< address_v6 >, 965
 ip::basic_resolver_iterator, 1019
 ip::basic_resolver_results, 1038
operator+=
 buffers_iterator, 750
 const_buffer, 769
 const_buffers_1, 772
 mutable_buffer, 1128
 mutable_buffers_1, 1131
operator-
 buffers_iterator, 750
operator--
 buffers_iterator, 751
 ip::basic_address_iterator< address_v4 >, 957
 ip::basic_address_iterator< address_v6 >, 966
operator-=
 buffers_iterator, 751
operator->
 buffers_iterator, 751
 ip::basic_address_iterator< address_v4 >, 957
 ip::basic_address_iterator< address_v6 >, 966
 ip::basic_resolver_iterator, 1020
 ip::basic_resolver_results, 1039
operator<
 buffers_iterator, 751
 generic::basic_endpoint, 837
 ip::address, 921
 ip::address_v4, 933
 ip::address_v6, 947
 ip::basic_endpoint, 983
 local::basic_endpoint, 1104
operator<<, 1135
 ip::address, 921
 ip::address_v4, 933
 ip::address_v6, 947
 ip::basic_endpoint, 984
 local::basic_endpoint, 1104
operator<=
 buffers_iterator, 752
 generic::basic_endpoint, 837
 ip::address, 921
 ip::address_v4, 934
 ip::address_v6, 948
 ip::basic_endpoint, 984
 local::basic_endpoint, 1104
operator=
 basic_datagram_socket, 325
 basic_deadline_timer, 358
 basic_io_object, 364
 basic_raw_socket, 411
 basic_seq_packet_socket, 474
 basic_socket, 527
 basic_socket_acceptor, 579
 basic_socket_iostream, 593
 basic_socket_streambuf, 601
 basic_stream_socket, 647
 basic_waitable_timer, 692
 executor, 820
 generic::basic_endpoint, 838
 ip::address, 922
 ip::address_v4, 934
 ip::address_v6, 949
 ip::basic_address_iterator< address_v4 >, 958
 ip::basic_address_iterator< address_v6 >, 966
 ip::basic_address_range< address_v4 >, 975
 ip::basic_address_range< address_v6 >, 979
 ip::basic_endpoint, 984
 ip::basic_resolver, 999
 ip::basic_resolver_iterator, 1020
 ip::basic_resolver_results, 1039
 ip::network_v4, 1060
 ip::network_v6, 1066
 local::basic_endpoint, 1104
 posix::descriptor, 1151
 posix::stream_descriptor, 1170
 serial_port, 1233
 ssl::context, 1282
 strand, 1334
 system_error, 1347
 windows::object_handle, 1385
 windows::overlapped_handle, 1393
 windows::random_access_handle, 1407
 windows::stream_handle, 1421

operator==
 buffers_iterator, 752
 error_category, 803
 error_code, 806
 executor, 821
 generic::basic_endpoint, 838
 generic::datagram_protocol, 843
 generic::raw_protocol, 850
 generic::seq_packet_protocol, 858
 generic::stream_protocol, 867
 io_context::executor_type, 900
 io_context::strand, 907
 ip::address, 922
 ip::address_v4, 935
 ip::address_v6, 949
 ip::basic_address_iterator< address_v4 >, 958
 ip::basic_address_iterator< address_v6 >, 966
 ip::basic_endpoint, 985
 ip::basic_resolver_iterator, 1020
 ip::basic_resolver_results, 1040
 ip::icmp, 1047
 ip::network_v4, 1061
 ip::network_v6, 1066
 ip::tcp, 1080
 ip::udp, 1089
 local::basic_endpoint, 1105
 strand, 1335
 system_executor, 1351
 thread_pool::executor_type, 1366

operator>
 buffers_iterator, 752
 generic::basic_endpoint, 838
 ip::address, 923
 ip::address_v4, 935
 ip::address_v6, 949
 ip::basic_endpoint, 985
 local::basic_endpoint, 1105

operator>=
 buffers_iterator, 752
 generic::basic_endpoint, 838
 ip::address, 923
 ip::address_v4, 935
 ip::address_v6, 949
 ip::basic_endpoint, 985
 local::basic_endpoint, 1105

operator[]
 basic_yield_context, 696
 buffers_iterator, 753
 use_future_t, 1373

options
 ssl::context, 1282
 ssl::context_base, 1303

out_of_band_inline
 basic_datagram_socket, 326
 basic_raw_socket, 412
 basic_seq_packet_socket, 475

basic_socket, 528
 basic_socket_acceptor, 580
 basic_stream_socket, 648
 socket_base, 1264

overflow
 basic_socket_streambuf, 601
 basic_streambuf, 670

overlapped_handle
 windows::overlapped_handle, 1394

overlapped_ptr
 windows::overlapped_ptr, 1396

owns_work
 executor_work_guard, 833

P

parity
 serial_port_base::parity, 1245

passive
 ip::basic_resolver, 999
 ip::basic_resolver_query, 1028
 ip::resolver_base, 1068
 ip::resolver_query_base, 1071

password_purpose
 ssl::context, 1283
 ssl::context_base, 1304

path
 local::basic_endpoint, 1105

peek
 buffered_read_stream, 726
 buffered_stream, 735
 buffered_write_stream, 743

pem
 ssl::context, 1279
 ssl::context_base, 1302

placeholders::bytes_transferred, 1135

placeholders::endpoint, 1135

placeholders::error, 1135

placeholders::iterator, 1136

placeholders::results, 1136

placeholders::signal_number, 1136

pointer
 buffers_iterator, 753
 ip::basic_address_iterator< address_v4 >, 958
 ip::basic_address_iterator< address_v6 >, 967
 ip::basic_resolver_iterator, 1020
 ip::basic_resolver_results, 1040

poll
 io_context, 887

poll_one
 io_context, 888

port
 ip::basic_endpoint, 986

post, 1177
 executor, 822
 io_context, 889
 io_context::executor_type, 900
 io_context::strand, 907

strand, 1335
system_executor, 1351
thread_pool::executor_type, 1366
prefix_length
 ip::network_v4, 1061
 ip::network_v6, 1066
prepare
 basic_streambuf, 670
 basic_streambuf_ref, 678
 dynamic_string_buffer, 790
 dynamic_vector_buffer, 796
protocol
 generic::basic_endpoint, 838
 generic::datagram_protocol, 843
 generic::raw_protocol, 850
 generic::seq_packet_protocol, 858
 generic::stream_protocol, 867
 ip::basic_endpoint, 986
 ip::icmp, 1047
 ip::tcp, 1080
 ip::udp, 1090
 local::basic_endpoint, 1106
 local::datagram_protocol, 1110
 local::stream_protocol, 1121
protocol_type
 basic_datagram_socket, 326
 basic_raw_socket, 413
 basic_seq_packet_socket, 475
 basic_socket, 528
 basic_socket_acceptor, 580
 basic_socket_iostream, 594
 basic_socket_streambuf, 601
 basic_stream_socket, 648
 generic::basic_endpoint, 839
 ip::basic_endpoint, 986
 ip::basic_resolver, 999
 ip::basic_resolver_entry, 1015
 ip::basic_resolver_query, 1028
 ip::basic_resolver_results, 1041
 local::basic_endpoint, 1106
puberror
 basic_socket_streambuf, 601

Q

query
 ip::basic_resolver, 1000

R

random_access_handle
 windows::random_access_handle, 1407

raw_protocol
 generic::raw_protocol, 850

rdbuf
 basic_socket_iostream, 594

read, 1178

read_at, 1191

read_some

basic_stream_socket, 649
buffered_read_stream, 727
buffered_stream, 736
buffered_write_stream, 744
posix::stream_descriptor, 1171
serial_port, 1233
ssl::stream, 1316
windows::stream_handle, 1422
read_some_at
 windows::random_access_handle, 1409
read_until, 1201
rebind
 use_future_t, 1373
receive
 basic_datagram_socket, 327
 basic_raw_socket, 413
 basic_seq_packet_socket, 476
 basic_stream_socket, 650
receive_buffer_size
 basic_datagram_socket, 329
 basic_raw_socket, 415
 basic_seq_packet_socket, 478
 basic_socket, 529
 basic_socket_acceptor, 580
 basic_stream_socket, 653
 socket_base, 1264
receive_from
 basic_datagram_socket, 330
 basic_raw_socket, 416
receive_low_watermark
 basic_datagram_socket, 332
 basic_raw_socket, 418
 basic_seq_packet_socket, 479
 basic_socket, 529
 basic_socket_acceptor, 581
 basic_stream_socket, 653
 socket_base, 1265
reference
 buffers_iterator, 753
 ip::basic_address_iterator< address_v4 >, 958
 ip::basic_address_iterator< address_v6 >, 967
 ip::basic_resolver_iterator, 1021
 ip::basic_resolver_results, 1041
release
 basic_datagram_socket, 332
 basic_raw_socket, 419
 basic_seq_packet_socket, 479
 basic_socket, 530
 basic_socket_acceptor, 582
 basic_stream_socket, 654
 posix::descriptor, 1151
 posix::stream_descriptor, 1172
 windows::overlapped_ptr, 1397
remote_endpoint
 basic_datagram_socket, 333
 basic_raw_socket, 420

basic_seq_packet_socket, 480
 basic_socket, 531
 basic_stream_socket, 655
 remove
 signal_set, 1254
 reserve
 basic_streambuf, 671
 reset
 executor_work_guard, 833
 io_context, 890
 windows::overlapped_ptr, 1397
 resize
 generic::basic_endpoint, 839
 ip::basic_endpoint, 986
 local::basic_endpoint, 1106
 resolve
 ip::basic_resolver, 1001
 resolver
 ip::icmp, 1047
 ip::tcp, 1080
 ip::udp, 1090
 resolver_errc::try_again, 1221
 restart
 io_context, 890
 result
 async_completion, 224
 result_type
 executor_binder, 830
 ssl::rfc2818_verification, 1306
 results_type
 ip::basic_resolver, 1010
 return_type
 async_result, 262
 async_result< std::packaged_task< Result(Args...) >, Signature >, 265
 reuse_address
 basic_datagram_socket, 335
 basic_raw_socket, 421
 basic_seq_packet_socket, 481
 basic_socket, 532
 basic_socket_acceptor, 583
 basic_stream_socket, 656
 socket_base, 1266
 rfc2818_verification
 ssl::rfc2818_verification, 1306
 run
 io_context, 890
 run_for
 io_context, 891
 run_one
 io_context, 892
 run_one_for
 io_context, 893
 run_one_until
 io_context, 893
 run_until

io_context, 894
 running_in_this_thread
 io_context::executor_type, 901
 io_context::strand, 908
 strand, 1335
 thread_pool::executor_type, 1367

S

scope_id
 ip::address_v6, 950
 second_argument_type
 executor_binder, 831
 send
 basic_datagram_socket, 335
 basic_raw_socket, 421
 basic_seq_packet_socket, 482
 basic_stream_socket, 657
 send_break
 serial_port, 1235
 send_buffer_size
 basic_datagram_socket, 337
 basic_raw_socket, 424
 basic_seq_packet_socket, 483
 basic_socket, 532
 basic_socket_acceptor, 583
 basic_stream_socket, 659
 socket_base, 1266
 send_low_watermark
 basic_datagram_socket, 338
 basic_raw_socket, 424
 basic_seq_packet_socket, 484
 basic_socket, 533
 basic_socket_acceptor, 584
 basic_stream_socket, 660
 socket_base, 1267
 send_to
 basic_datagram_socket, 339
 basic_raw_socket, 425
 seq_packet_protocol
 generic::seq_packet_protocol, 858
 serial_port
 serial_port, 1236
 server
 ssl::stream, 1314
 ssl::stream_base, 1325
 service
 execution_context::service, 815
 io_context::service, 902
 service_already_exists
 service_already_exists, 1247
 service_name
 ip::basic_resolver_entry, 1015
 ip::basic_resolver_query, 1028
 service_not_found
 error::addrinfo_errors, 797
 service_type
 basic_io_object, 364

set_default_verify_paths
 ssl::context, 1283
 set_option
 basic_datagram_socket, 341
 basic_raw_socket, 427
 basic_seq_packet_socket, 485
 basic_socket, 534
 basic_socket_acceptor, 584
 basic_stream_socket, 660
 serial_port, 1238
 set_options
 ssl::context, 1284
 set_password_callback
 ssl::context, 1285
 set_verify_callback
 ssl::context, 1286
 ssl::stream, 1318
 set_verify_depth
 ssl::context, 1287
 ssl::stream, 1319
 set_verify_mode
 ssl::context, 1288
 ssl::stream, 1320
 setbuf
 basic_socket_streambuf, 601
 shut_down
 error::basic_errors, 797
 shutdown
 basic_datagram_socket, 342
 basic_raw_socket, 429
 basic_seq_packet_socket, 486
 basic_socket, 535
 basic_stream_socket, 662
 execution_context, 812
 execution_context::service, 816
 io_context, 894
 ssl::stream, 1321
 system_context, 1345
 thread_pool, 1361
 shutdown_both
 basic_datagram_socket, 344
 basic_raw_socket, 430
 basic_seq_packet_socket, 487
 basic_socket, 536
 basic_socket_acceptor, 586
 basic_stream_socket, 663
 socket_base, 1267
 shutdown_receive
 basic_datagram_socket, 344
 basic_raw_socket, 430
 basic_seq_packet_socket, 487
 basic_socket, 536
 basic_socket_acceptor, 586
 basic_stream_socket, 663
 socket_base, 1267
 shutdown_send

 basic_datagram_socket, 344
 basic_raw_socket, 430
 basic_seq_packet_socket, 487
 basic_socket, 536
 basic_socket_acceptor, 586
 basic_stream_socket, 663
 socket_base, 1267
 shutdown_type
 basic_datagram_socket, 344
 basic_raw_socket, 430
 basic_seq_packet_socket, 487
 basic_socket, 536
 basic_socket_acceptor, 586
 basic_stream_socket, 663
 socket_base, 1267
 signal_set
 signal_set, 1255
 single_dh_use
 ssl::context, 1289
 ssl::context_base, 1304
 size
 basic_streambuf, 671
 basic_streambuf_ref, 678
 const_buffer, 769
 const_buffers_1, 772
 dynamic_string_buffer, 791
 dynamic_vector_buffer, 796
 generic::basic_endpoint, 839
 ip::basic_address_range<address_v4>, 975
 ip::basic_endpoint, 987
 ip::basic_resolver_results, 1042
 local::basic_endpoint, 1106
 mutable_buffer, 1129
 mutable_buffers_1, 1132
 size_type
 ip::basic_resolver_results, 1042
 socket
 basic_socket_iostream, 594
 basic_socket_streambuf, 602
 generic::datagram_protocol, 843
 generic::raw_protocol, 851
 generic::seq_packet_protocol, 859
 generic::stream_protocol, 867
 ip::icmp, 1049
 ip::tcp, 1082
 ip::udp, 1092
 local::datagram_protocol, 1110
 local::stream_protocol, 1121
 socket_type_not_supported
 error::addrinfo_errors, 797
 software
 serial_port_base::flow_control, 1244
 spawn, 1268
 ssl
 ssl::stream::impl_struct, 1324
 ssl::error::get_stream_category, 1304

ssl::error::make_error_code, 1304
 ssl::error::stream_category, 1304
 ssl::error::stream_errors, 1305
 ssl::verify_client_once, 1325
 ssl::verify_fail_if_no_peer_cert, 1326
 ssl::verify_mode, 1327
 ssl::verify_none, 1327
 ssl::verify_peer, 1327
 sslv2
 ssl::context, 1280
 ssl::context_base, 1302
 sslv23
 ssl::context, 1280
 ssl::context_base, 1302
 sslv23_client
 ssl::context, 1280
 ssl::context_base, 1302
 sslv23_server
 ssl::context, 1280
 ssl::context_base, 1302
 sslv2_client
 ssl::context, 1280
 ssl::context_base, 1302
 sslv2_server
 ssl::context, 1280
 ssl::context_base, 1302
 sslv3
 ssl::context, 1280
 ssl::context_base, 1302
 sslv3_client
 ssl::context, 1280
 ssl::context_base, 1302
 sslv3_server
 ssl::context, 1280
 ssl::context_base, 1302
 steady_timer, 1327
 stop
 io_context, 894
 system_context, 1345
 thread_pool, 1361
 stop_bits
 serial_port_base::stop_bits, 1246
 stopped
 io_context, 894
 system_context, 1345
 store
 serial_port_base::baud_rate, 1242
 serial_port_base::character_size, 1243
 serial_port_base::flow_control, 1244
 serial_port_base::parity, 1245
 serial_port_base::stop_bits, 1246
 strand
 io_context::strand, 908
 strand, 1336
 stream
 ssl::stream, 1322
 stream_descriptor
 posix::stream_descriptor, 1172
 stream_handle
 windows::stream_handle, 1423
 stream_protocol
 generic::stream_protocol, 871
 stream_truncated
 ssl::error::stream_errors, 1305
 streambuf, 1337
 subtract
 time_traits< boost::posix_time::ptime >, 1368
 swap
 ip::basic_resolver_results, 1042
 sync
 basic_socket_streambuf, 602
 system_category, 1339
 system_error
 system_error, 1347
 system_timer, 1351

T

target
 executor, 822
 target_type
 executor, 823
 executor_binder, 831
 thread
 thread, 1355
 thread_pool
 thread_pool, 1361
 time_point
 basic_socket_iostream, 594
 basic_socket_streambuf, 602
 basic_waitable_timer, 692
 time_type
 basic_deadline_timer, 358
 basic_socket_iostream, 594
 basic_socket_streambuf, 602
 time_traits< boost::posix_time::ptime >, 1368
 timed_out
 error::basic_errors, 797
 tls
 ssl::context, 1280
 ssl::context_base, 1302
 tls_client
 ssl::context, 1280
 ssl::context_base, 1302
 tls_server
 ssl::context, 1280
 ssl::context_base, 1302
 tlsv1
 ssl::context, 1280
 ssl::context_base, 1302
 tlsv11
 ssl::context, 1280
 ssl::context_base, 1302
 tlsv11_client

ssl::context, 1280
ssl::context_base, 1302
tlsv11_server
 ssl::context, 1280
 ssl::context_base, 1302
tlsv12
 ssl::context, 1280
 ssl::context_base, 1302
tlsv12_client
 ssl::context, 1280
 ssl::context_base, 1302
tlsv12_server
 ssl::context, 1280
 ssl::context_base, 1302
tlsv1_client
 ssl::context, 1280
 ssl::context_base, 1302
tlsv1_server
 ssl::context, 1280
 ssl::context_base, 1302
to_bytes
 ip::address_v4, 935
 ip::address_v6, 950
to_posix_duration
 time_traits< boost::posix_time::ptime >, 1369
to_string
 ip::address, 923
 ip::address_v4, 936
 ip::address_v6, 950
 ip::network_v4, 1061
 ip::network_v6, 1066
to_uint
 ip::address_v4, 936
to_ulong
 ip::address_v4, 936
to_v4
 ip::address, 924
 ip::address_v6, 951
to_v6
 ip::address, 924
to_wait_duration
 wait_traits, 1375
traits_type
 basic_deadline_timer, 358
 basic_waitable_timer, 692
transfer_all, 1369
transfer_at_least, 1369
transfer_exactly, 1370
try_again
 error::basic_errors, 797
two
 serial_port_base::stop_bits, 1246
type
 associated_allocator, 220
 associated_executor, 222
 async_result< Handler >, 264
generic::datagram_protocol, 847
generic::raw_protocol, 855
generic::seq_packet_protocol, 862
generic::stream_protocol, 871
handler_type, 874
ip::icmp, 1053
ip::tcp, 1086
ip::udp, 1095
local::datagram_protocol, 1114
local::stream_protocol, 1124
serial_port_base::flow_control, 1244
serial_port_base::parity, 1245
serial_port_base::stop_bits, 1246

U

uint_type
 ip::address_v4, 936
underflow
 basic_socket_streambuf, 602
 basic_streambuf, 671
unspecified_bool_true
 error_code, 806
 executor, 823
unspecified_bool_type
 error_code, 806
 executor, 823
use_certificate
 ssl::context, 1290
use_certificate_chain
 ssl::context, 1291
use_certificate_chain_file
 ssl::context, 1292
use_certificate_file
 ssl::context, 1293
use_future, 1371
use_future_t
 use_future_t, 1373
use_private_key
 ssl::context, 1294
use_private_key_file
 ssl::context, 1295
use_rsa_private_key
 ssl::context, 1296
use_rsa_private_key_file
 ssl::context, 1297
use_service, 1373
 execution_context, 812
 io_context, 895
 system_context, 1345
 thread_pool, 1362
use_tmp_dh
 ssl::context, 1298
use_tmp_dh_file
 ssl::context, 1299

V

V
V4

ip::icmp, 1053
 ip::tcp, 1086
 ip::udp, 1095
v4_compatible
 ip::address_v6, 951
v4_mapped
 ip::address_v6, 951
 ip::basic_resolver, 1012
 ip::basic_resolver_query, 1028
 ip::resolver_base, 1069
 ip::resolver_query_base, 1071
 ip::v4_mapped_t, 1096
v6
 ip::icmp, 1053
 ip::tcp, 1086
 ip::udp, 1095
value
 error_code, 807
 is_endpoint_sequence, 1098
 is_match_condition, 1099
 is_read_buffered, 1099
 is_write_buffered, 1100
 serial_port_base::baud_rate, 1242
 serial_port_base::character_size, 1243
 serial_port_base::flow_control, 1244
 serial_port_base::parity, 1245
 serial_port_base::stop_bits, 1247
value_type
 buffers_iterator, 753
 const_buffers_1, 772
 ip::basic_address_iterator< address_v4 >, 960
 ip::basic_address_iterator< address_v6 >, 969
 ip::basic_resolver_iterator, 1022
 ip::basic_resolver_results, 1042
 mutable_buffers_1, 1132
 null_buffers, 1134
values_
 ip::basic_resolver_iterator, 1022
 ip::basic_resolver_results, 1043
verify_context
 ssl::verify_context, 1326

W

wait
 basic_datagram_socket, 344
 basic_deadline_timer, 359
 basic_raw_socket, 430
 basic_seq_packet_socket, 488
 basic_socket, 536
 basic_socket_acceptor, 586
 basic_stream_socket, 663
 basic_waitable_timer, 693
 posix::descriptor, 1152
 posix::stream_descriptor, 1174
 windows::object_handle, 1385
wait_error
 basic_datagram_socket, 345

basic_raw_socket, 431
 basic_seq_packet_socket, 489
 basic_socket, 537
 basic_socket_acceptor, 587
 basic_stream_socket, 664
 posix::descriptor, 1153
 posix::descriptor_base, 1154
 posix::stream_descriptor, 1175
 socket_base, 1267
wait_read
 basic_datagram_socket, 345
 basic_raw_socket, 431
 basic_seq_packet_socket, 489
 basic_socket, 537
 basic_socket_acceptor, 587
 basic_stream_socket, 664
 posix::descriptor, 1153
 posix::descriptor_base, 1154
 posix::stream_descriptor, 1175
 socket_base, 1267
wait_type
 basic_datagram_socket, 345
 basic_raw_socket, 431
 basic_seq_packet_socket, 489
 basic_socket, 537
 basic_socket_acceptor, 587
 basic_stream_socket, 664
 posix::descriptor, 1153
 posix::descriptor_base, 1154
 posix::stream_descriptor, 1175
 socket_base, 1267
wait_write
 basic_datagram_socket, 345
 basic_raw_socket, 431
 basic_seq_packet_socket, 489
 basic_socket, 537
 basic_socket_acceptor, 587
 basic_stream_socket, 664
 posix::descriptor, 1153
 posix::descriptor_base, 1154
 posix::stream_descriptor, 1175
 socket_base, 1267
what
 bad_executor, 278
 ip::bad_address_cast, 954
 system_error, 1348
work
 io_context::work, 910
would_block
 error::basic_errors, 797
wrap
 io_context, 896
 io_context::strand, 908
write, 1426
write_at, 1439
write_some

basic_stream_socket, 665
buffered_read_stream, 727
buffered_stream, 736
buffered_write_stream, 745
posix::stream_descriptor, 1175
serial_port, 1239
ssl::stream, 1322
windows::stream_handle, 1425
write_some_at
 windows::random_access_handle, 1410

Y

yield_context, 1448