PD ISO/IEC/TS 19216:2018

BSI Standards Publication

Programming Languages — C++ Extensions for Networking

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

National foreword

This Published Document is the UK implementation of ISO/IEC TS 19216:2018.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, their environments and system software interfaces.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2018
Published by BSI Standards Limited 2018

ISBN 978 0 580 97077 1

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This Published Document was published under the authority of the Standards Policy and Strategy Committee on 31 May 2018.

Amendments/corrigenda issued since publication

Date	Text affected

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

TECHNICAL SPECIFICATION

ISO/IEC TS 19216

First edition
2018-04

Programming Languages — C++ Extensions for Networking

Langages de programmation — Extensions C++ pour mise en réseau

Reference number
ISO/IEC TS 19216:2018(E)

Documents
Document Sharing Hub

© ISO/IEC 2018

Full document details: PD ISO/IEC TS 19216:2018

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2018

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

© ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

Contents

Foreword vi

1 Scope 1

2 Normative references 2

3 Terms and definitions 3

4 General Principles 4
4.1 Conformance . . . 4
4.2 Acknowledgments . . . 4

5 Namespaces and headers 5

6 Future plans (Informative) 6

7 Feature test macros (Informative) 7

8 Method of description (Informative) 8
8.1 Structure of each clause . . . 8
8.2 Other conventions . . . 8

9 Error reporting 9
9.1 Synchronous operations . . . 9
9.2 Asynchronous operations . . . 10
9.3 Error conditions . . . 10
9.4 Suppression of signals . . . 10

10 Library summary 11

11 Convenience header 13
11.1 Header `<experimental/net>` synopsis . . . 13

12 Forward declarations 14
12.1 Header `<experimental/netfwd>` synopsis . . . 14

13 Asynchronous model 16
13.1 Header `<experimental/executor>` synopsis . . . 16
13.2 Requirements . . . 19
13.3 Class template `async_result` . . . 27
13.4 Class template `async_completion` . . . 28
13.5 Class template `associated_allocator` . . . 29
13.6 Function `get_associated_allocator` . . . 30
13.7 Class `execution_context` . . . 30
13.8 Class `execution_context::service` . . . 32
13.9 Class template `is_executor` . . . 33
13.10 Executor argument tag . . . 33
13.11 `uses_executor` . . . 34

© ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

13.12 Class template `associated_executor` . . . 34
13.13 Function `get_associated_executor` . . . 35
13.14 Class template `executor_binder` . . . 36
13.15 Function `bind_executor` . . . 39
13.16 Class template `executor_work_guard` . . . 40
13.17 Function `make_work_guard` . . . 41
13.18 Class `system_executor` . . . 42
13.19 Class `system_context` . . . 43
13.20 Class `bad_executor` . . . 44
13.21 Class `executor` . . . 45
13.22 Function `dispatch` . . . 49
13.23 Function `post` . . . 50
13.24 Function `defer` . . . 51
13.25 Class template `strand` . . . 52
13.26 Class template `use_future_t` . . . 56
13.27 Partial specialization of `async_result` for `packaged_task` . . . 59

14 Basic I/O services **61**
14.1 Header `<experimental/io_context>` synopsis . . . 61
14.2 Class `io_context` . . . 61
14.3 Class `io_context::executor_type` . . . 65

15 Timers **67**
15.1 Header `<experimental/timer>` synopsis . . . 67
15.2 Requirements . . . 67
15.3 Class template `wait_traits` . . . 68
15.4 Class template `basic_waitable_timer` . . . 69

16 Buffers **73**
16.1 Header `<experimental/buffer>` synopsis . . . 73
16.2 Requirements . . . 78
16.3 Error codes . . . 82
16.4 Class `mutable_buffer` . . . 82
16.5 Class `const_buffer` . . . 83
16.6 Buffer type traits . . . 84
16.7 Buffer sequence access . . . 85
16.8 Function `buffer_size` . . . 85
16.9 Function `buffer_copy` . . . 85
16.10 Buffer arithmetic . . . 86
16.11 Buffer creation functions . . . 86
16.12 Class template `dynamic_vector_buffer` . . . 88
16.13 Class template `dynamic_string_buffer` . . . 89
16.14 Dynamic buffer creation functions . . . 91

17 Buffer-oriented streams **92**
17.1 Requirements . . . 92
17.2 Class `transfer_all` . . . 94
17.3 Class `transfer_at_least` . . . 95
17.4 Class `transfer_exactly` . . . 95
17.5 Synchronous read operations . . . 96
17.6 Asynchronous read operations . . . 98

© ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

17.7 Synchronous write operations . . . 99
17.8 Asynchronous write operations . . . 100
17.9 Synchronous delimited read operations . . . 102
17.10 Asynchronous delimited read operations . . . 102

18 Sockets **104**
18.1 Header `<experimental/socket>` synopsis . . . 104
18.2 Requirements . . . 106
18.3 Error codes . . . 115
18.4 Class `socket_base` . . . 116
18.5 Socket options . . . 118
18.6 Class template `basic_socket` . . . 121
18.7 Class template `basic_datagram_socket` . . . 131
18.8 Class template `basic_stream_socket` . . . 139
18.9 Class template `basic_socket_acceptor` . . . 145

19 Socket iostreams **157**
19.1 Class template `basic_socket_streambuf` . . . 157
19.2 Class template `basic_socket_iostream` . . . 161

20 Socket algorithms **164**
20.1 Synchronous connect operations . . . 164
20.2 Asynchronous connect operations . . . 165

21 Internet protocol **167**
21.1 Header `<experimental/internet>` synopsis . . . 167
21.2 Requirements . . . 171
21.3 Error codes . . . 173
21.4 Class `ip::address` . . . 174
21.5 Class `ip::address_v4` . . . 177
21.6 Class `ip::address_v6` . . . 181
21.7 Class `ip::bad_address_cast` . . . 186
21.8 Hash support . . . 187
21.9 Class template `ip::basic_address_iterator` specializations . . . 187
21.10 Class template `ip::basic_address_range` specializations . . . 188
21.11 Class template `ip::network_v4` . . . 190
21.12 Class template `ip::network_v6` . . . 193
21.13 Class template `ip::basic_endpoint` . . . 195
21.14 Class template `ip::basic_resolver_entry` . . . 199
21.15 Class template `ip::basic_resolver_results` . . . 201
21.16 Class `ip::resolver_base` . . . 204
21.17 Class template `ip::basic_resolver` . . . 205
21.18 Host name functions . . . 211
21.19 Class `ip::tcp` . . . 211
21.20 Class `ip::udp` . . . 212
21.21 Internet socket options . . . 214

Index **219**

Index of library names **221**

Index of implementation-defined behavior **227**

© ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces.*

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

1 Scope [scope]

1 This document describes extensions to the C++ Standard Library. This document specifies requirements for implementations of an interface that computer programs written in the C++ programming language may use to perform operations related to networking, such as operations involving sockets, timers, buffer management, host name resolution and internet protocols. This document is applicable to information technology systems that can perform network operations, such as those with operating systems that conform to the POSIX interface. This document is applicable only to vendors who wish to provide the interface it describes.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

2 Normative references [references]

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2014, *Programming languages — C++*

(1.2) — ISO/IEC TS 19568:2015, *C++ Extensions for Library Fundamentals*

(1.3) — ISO/IEC 9945:2009, *Information Technology — Portable Operating System Interface (POSIX)*

(1.4) — ISO/IEC 2382-1:1993, *Information technology — Vocabulary*

2 The programming language and library described in ISO/IEC 14882 is herein called the C++ Standard. References to clauses within the C++ Standard are written as "C++ 2014, Clause 17". The operating system interface described in ISO/IEC 9945 is herein called POSIX.

3 This document mentions commercially available operating systems for purposes of exposition. POSIX® is a registered trademark of The IEEE. Windows® is a registered trademark of Microsoft Corporation. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of these products.

4 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++ 2014, Clause 17) is included into this document by reference.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

3 Terms and definitions [defs]

1 For the purposes of this document, the terms and definitions given in ISO/IEC 14882:2014, ISO/IEC 2382-1:1993, and the following apply.

2 ISO and IEC maintain terminological databases for use in standardization at the following addresses:

(2.1) — IEC Electropedia: available at http://www.electropedia.org/

(2.2) — ISO Online browsing platform: available at http://www.iso.org/obp

3 Terms that are used only in a small portion of this document are defined where they are used and italicized where they are defined.

3.1 **[defs.host.byte.order]**
host byte order
the arrangement of bytes in any integer type when using a specific machine architecture

[SOURCE: ISO/IEC 9945:2009, 3.193]

3.2 **[defs.net.byte.order]**
network byte order
the way of representing any integer type such that, when transmitted over a network via a network endpoint, the `int` type is transmitted as an appropriate number of octets with the most significant octet first, followed by any other octets in descending order of significance

[SOURCE: ISO/IEC 9945:2009, 3.237]

3.3 **[defs.sync.op]**
synchronous operation
operation where control is not returned until the operation completes

3.4 **[defs.async.op]**
asynchronous operation
operation where control is returned immediately without waiting for the operation to complete

[*Note 1 to entry:* Multiple asynchronous operations may be executed concurrently. — *end note*]

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

4 General Principles [general]

4.1 Conformance [conformance]

1 Conformance is specified in terms of behavior. Ideal behavior is not always implementable, so the conformance subclauses take that into account.

4.1.1 POSIX conformance [conformance.9945]

1 Some behavior is specified by reference to POSIX. How such behavior is actually implemented is unspecified.

2 [*Note:* This constitutes an "as if" rule allowing implementations to call native operating system or other APIs. — *end note*]

3 Implementations are encouraged to provide such behavior as it is defined by POSIX. Implementations shall document any behavior that differs from the behavior defined by POSIX. Implementations that do not support exact POSIX behavior are encouraged to provide behavior as close to POSIX behavior as is reasonable given the limitations of actual operating systems and file systems. If an implementation cannot provide any reasonable behavior, the implementation shall report an error as specified in Error Reporting (9).

4 [*Note:* This allows users to rely on an exception being thrown or an error code being set when an implementation cannot provide any reasonable behavior. — *end note*]

5 Implementations are not required to provide behavior that is not supported by a particular operating system.

4.1.2 Conditionally-supported features [conformance.conditional]

1 This document defines conditionally-supported features, in the form of additional member functions on types that satisfy `Protocol` (18.2.6), `Endpoint` (18.2.4), `SettableSocketOption` (18.2.9), `GettableSocketOption` (18.2.8) or `IoControlCommand` (18.2.12) requirements.

2 [*Note:* This is so that, when the additional member functions are available, C++ programs can extend the library to add support for other protocols and socket options. — *end note*]

3 For the purposes of this document, implementations that provide all of the additional member functions are known as extensible implementations.

4 [*Note:* Implementations are encouraged to provide the additional member functions, where possible. It is intended that POSIX and Windows implementations will provide them. — *end note*]

4.2 Acknowledgments [intro.ack]

1 The design of this specification is based, in part, on the Asio library written by Christopher Kohlhoff.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

5 Namespaces and headers [namespaces]

1 The components described in this document are experimental and not part of the C++ standard library. All components described in this document are declared in namespace `std::experimental::net::v1` or a sub-namespace thereof unless otherwise specified. The headers described in this document shall import the contents of `std::experimental::net::v1` into `std::experimental::net` as if by:

```
namespace std {
  namespace experimental {
    namespace net {
      inline namespace v1 {}
    }
  }
}
```

2 Unless otherwise specified, references to other entities described in this document are assumed to be qualified with `std::experimental::net::v1::`, references to entities described in the C++ standard are assumed to be qualified with `std::`, and references to entities described in C++ Extensions for Library Fundamentals are assumed to be qualified with `std::experimental::fundamentals_v2::`.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

6 Future plans (Informative) [plans]

1 This clause describes tentative plans for future versions of this document and plans for moving content into future versions of the C++ Standard.

2 The C++ committee may release new versions of this document, containing networking library extensions we hope to add to a near-future version of the C++ Standard. Future versions will define their contents in `std::experimental::net::v2`, `std::experimental::net::v3`, etc., with the most recent implemented version inlined into `std::experimental::net`.

3 When an extension defined in this or a future version of this document represents enough existing practice, it will be moved into the next version of the C++ Standard by replacing the `experimental::net::v`*N* segment of its namespace with `net`, and by removing the `experimental/` prefix from its header's path.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

7 Feature test macros (Informative) [feature.test]

1 These macros allow users to determine which version of this document is supported by the headers defined by the specification. All headers in this document shall define the `__cpp_lib_experimental_net` feature test macro in Table 1.

2 If an implementation supplies all of the conditionally-supported features specified in 4.1.2, all headers in this document shall additionally define the `__cpp_lib_experimental_net_extensible` feature test macro.

Table 1 — Feature-test macro(s)

Macro name	Value
`__cpp_lib_experimental_net`	201707
`__cpp_lib_experimental_net_extensible`	201707

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TS 19216:2018

8 Method of description (Informative) [description]

1 This subclause describes the conventions used to specify this document, in addition to those conventions specified in C++ 2014, 17.5.

8.1 Structure of each clause [structure]

8.1.1 Detailed specifications [structure.specifications]

1 In addition to the elements defined in C++ 2014, 17.5.1.4, descriptions of function semantics contain the following elements (as appropriate):

(1.1) — *Completion signature:* if the function initiates an asynchronous operation, specifies the signature of a completion handler used to receive the result of the operation.

8.2 Other conventions [conventions]

8.2.1 Nested classes [nested.class]

1 Several classes defined in this document are nested classes. For a specified nested class `A::B`, an implementation is permitted to define `A::B` as a synonym for a class with equivalent functionality to class `A::B`. [*Note:* When `A::B` is a synonym for another type `A` provides a nested type `B`, to emulate the injected class name. — *end note*]

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

9 Error reporting [err.report]

9.1 Synchronous operations [err.report.sync]

1 Most synchronous network library functions provide two overloads, one that throws an exception to report system errors, and another that sets an `error_code` (C++ 2014, 19.5).

[*Note:* This supports two common use cases:

(1.1) — Uses where system errors are truly exceptional and indicate a serious failure. Throwing an exception is the most appropriate response.

(1.2) — Uses where system errors are routine and do not necessarily represent failure. Returning an error code is the most appropriate response. This allows application specific error handling, including simply ignoring the error.

— *end note*]

2 Functions not having an argument of type `error_code&` report errors as follows, unless otherwise specified:

(2.1) — When a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, the function exits via an exception of a type that would match a handler of type `system_error`.

(2.2) — Destructors throw nothing.

3 Functions having an argument of type `error_code&` report errors as follows, unless otherwise specified:

(3.1) — If a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, the `error_code&` argument `ec` is set as appropriate for the specific error. Otherwise, the `ec` argument is set such that `!ec` is `true`.

4 Where a function is 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);
```

5 then, when R is non-`void`, the effects of the first overload are as if:

```
error_code ec;
R r(f(a1, a2, ..., aN, ec));
if (ec) throw system_error(ec, S);
return r;
```

6 otherwise, when R is `void`, the effects of the first overload are as if:

```
error_code ec;
f(a1, a2, ..., aN, ec);
if (ec) throw system_error(ec, S);
```

7 except that the type thrown may differ as specified above. `S` is an NTBS indicating where the exception was thrown. [*Note:* A possible value for `S` is `__func__`. — *end note*]

8 For both overloads, failure to allocate storage is reported by throwing an exception as described in the C++ standard (C++ 2014, 17.6.5.12).

9 In this document, when a type requirement is specified using two function call expressions *f*, with and without an argument `ec` of type `error_code`:

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

```
f(a1, a2, ..., aN)
f(a1, a2, ..., aN, ec)
```

then the effects of the first call expression of *f* shall be as described for the first overload above.

9.2 Asynchronous operations [err.report.async]

1 Asynchronous network library functions in this document are identified by having the prefix `async_` and take a completion handler (13.2.7.2). These asynchronous operations report errors as follows:

(1.1) — If a call by the implementation to an operating system or other underlying API results in an error that prevents the asynchronous operation from meeting its specifications, the completion handler is invoked with an `error_code` value `ec` that is set as appropriate for the specific error. Otherwise, the `error_code` value `ec` is set such that `!ec` is `true`.

(1.2) — Asynchronous operations shall not fail with an error condition that indicates interruption of an operating system or underlying API by a signal. [*Note:* Such as POSIX error number `EINTR` — *end note*] Asynchronous operations shall not fail with any error condition associated with non-blocking operations. [*Note:* Such as POSIX error numbers `EWOULDBLOCK`, `EAGAIN`, or `EINPROGRESS`; Windows error numbers `WSAEWOULDBLOCK` or `WSAEINPROGRESS` — *end note*]

2 In this document, when a type requirement is specified as a call to a function or member function having the prefix `async_`, then the function shall satisfy the error reporting requirements described above.

9.3 Error conditions [err.report.conditions]

1 Unless otherwise specified, when the behavior of a synchronous or asynchronous operation is defined "as if" implemented by a POSIX function, the `error_code` produced by the function shall meet the following requirements:

(1.1) — If the failure condition is one that is listed by POSIX for that function, the `error_code` shall compare equal to the error's corresponding `enum class errc` (C++ 2014, 19.5) or `enum class resolver_errc` (21.3) constant.

(1.2) — Otherwise, the `error_code` shall be set to an implementation-defined value that reflects the underlying operating system error.

2 [*Example:* The POSIX specification for `shutdown` lists `EBADF` as one of its possible errors. If a function that is specified "as if" implemented by `shutdown` fails with `EBADF` then the following condition holds for the `error_code` value `ec`: `ec == errc::bad_file_descriptor` — *end example*]

3 When the description of a function contains the element Error conditions, this lists conditions where the operation may fail. The conditions are listed, together with a suitable explanation, as `enum class` constants. Unless otherwise specified, this list is a subset of the failure conditions associated with the function.

9.4 Suppression of signals [err.report.signal]

1 Some POSIX functions referred to in this document may report errors by raising a `SIGPIPE` signal. Where a synchronous or asynchronous operation is specified in terms of these POSIX functions, the generation of `SIGPIPE` is suppressed and an error condition corresponding to POSIX `EPIPE` is produced instead.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

10 Library summary [summary]

Table 2 — Networking library summary

Clause	Header(s)
Convenience header (11)	`<experimental/net>`
Forward declarations (12)	`<experimental/netfwd>`
Asynchronous model (13)	`<experimental/executor>`
Basic I/O services (14)	`<experimental/io_context>`
Timers (15)	`<experimental/timer>`
Buffers (16) Buffer-oriented streams (17)	`<experimental/buffer>`
Sockets (18) Socket iostreams (19) Socket algorithms (20)	`<experimental/socket>`
Internet protocol (21)	`<experimental/internet>`

1 Throughout this document, the names of the template parameters are used to express type requirements, as listed in Table 3.

Table 3 — Template parameters and type requirements

template parameter name	type requirements
`AcceptableProtocol`	acceptable protocol (18.2.7)
`Allocator`	C++ 2014, 17.6.3.5
`AsyncReadStream`	buffer-oriented asynchronous read stream (17.1.2)
`AsyncWriteStream`	buffer-oriented asynchronous write stream (17.1.4)
`Clock`	C++ 2014, 20.12.3
`CompletionCondition`	completion condition (17.1.5)
`CompletionToken`	completion token (13.2.7.2)
`ConnectCondition`	connect condition (18.2.13)
`ConstBufferSequence`	constant buffer sequence (16.2.2)
`DynamicBuffer`	dynamic buffer (16.2.4)
`EndpointSequence`	endpoint sequence (18.2.5)
`ExecutionContext`	execution context (13.2.3)
`Executor`	executor (13.2.2)
`GettableSocketOption`	gettable socket option (18.2.8)
`InternetProtocol`	Internet protocol (21.2.1)
`IoControlCommand`	I/O control command (18.2.12)
`MutableBufferSequence`	mutable buffer sequence (16.2.1)
`ProtoAllocator`	proto-allocator (13.2.1)
`Protocol`	protocol (18.2.6)
`Service`	service (13.2.4)
`SettableSocketOption`	settable socket option (18.2.9)
`Signature`	signature (13.2.5)

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

Table 3 — Template parameters and type requirements (continued)

template parameter name	type requirements
`SyncReadStream`	buffer-oriented synchronous read stream (17.1.1)
`SyncWriteStream`	buffer-oriented synchronous write stream (17.1.3)
`WaitTraits`	wait traits (15.2.1)

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

11 Convenience header [convenience.hdr]

11.1 Header `<experimental/net>` synopsis [convenience.hdr.synop]

```
#include <experimental/executor>
#include <experimental/io_context>
#include <experimental/timer>
#include <experimental/buffer>
#include <experimental/socket>
#include <experimental/internet>
```

1 [*Note:* This header is provided as a convenience for programs so that they may access all networking facilities via a single, self-contained `#include`. — *end note*]

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

12 Forward declarations [fwd.decl]

12.1 Header <experimental/netfwd> synopsis [fwd.decl.synop]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class execution_context;
  template<class T, class Executor>
    class executor_binder;
  template<class Executor>
    class executor_work_guard;
  class system_executor;
  class executor;
  template<class Executor>
    class strand;

  class io_context;

  template<class Clock> struct wait_traits;
  template<class Clock, class WaitTraits = wait_traits<Clock>>
    class basic_waitable_timer;
  using system_timer = basic_waitable_timer<chrono::system_clock>;
  using steady_timer = basic_waitable_timer<chrono::steady_clock>;
  using high_resolution_timer = basic_waitable_timer<chrono::high_resolution_clock>;

  template<class Protocol>
    class basic_socket;
  template<class Protocol>
    class basic_datagram_socket;
  template<class Protocol>
    class basic_stream_socket;
  template<class Protocol>
    class basic_socket_acceptor;
  template<class Protocol, class Clock = chrono::steady_clock,
    class WaitTraits = wait_traits<Clock>>
      class basic_socket_streambuf;
  template<class Protocol, class Clock = chrono::steady_clock,
    class WaitTraits = wait_traits<Clock>>
      class basic_socket_iostream;

  namespace ip {

    class address;
    class address_v4;
    class address_v6;
    template<class Address>
      class basic_address_iterator;
    using address_v4_iterator = basic_address_iterator<address_v4>;
    using address_v6_iterator = basic_address_iterator<address_v6>;
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

```
    template<class Address>
      class basic_address_range;
    using address_v4_range = basic_address_range<address_v4>;
    using address_v6_range = basic_address_range<address_v6>;
    class network_v4;
    class network_v6;
    template<class InternetProtocol>
      class basic_endpoint;
    template<class InternetProtocol>
      class basic_resolver_entry;
    template<class InternetProtocol>
      class basic_resolver_results;
    template<class InternetProtocol>
      class basic_resolver;
    class tcp;
    class udp;

  } // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 Default template arguments are described as appearing both in `<netfwd>` and in the synopsis of other headers but it is well-formed to include both `<netfwd>` and one or more of the other headers. [*Note:* It is the implementation's responsibility to implement headers so that including `<netfwd>` and other headers does not violate the rules about multiple occurrences of default arguments. — *end note*]

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

13 Asynchronous model [async]

13.1 Header <experimental/executor> synopsis [async.synop]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class CompletionToken, class Signature>
    class async_result;

  template<class CompletionToken, class Signature>
    struct async_completion;

  template<class T, class ProtoAllocator = allocator<void>>
    struct associated_allocator;

  template<class T, class ProtoAllocator = allocator<void>>
    using associated_allocator_t = typename associated_allocator<T, ProtoAllocator>::type;

  // 13.6, get_associated_allocator:

  template<class T>
    associated_allocator_t<T> get_associated_allocator(const T& t) noexcept;
  template<class T, class ProtoAllocator>
    associated_allocator_t<T, ProtoAllocator>
      get_associated_allocator(const T& t, const ProtoAllocator& a) noexcept;

  enum class fork_event {
    prepare,
    parent,
    child
  };

  class execution_context;

  class service_already_exists;

  template<class Service> Service& use_service(execution_context& ctx);
  template<class Service, class... Args> Service&
    make_service(execution_context& ctx, Args&&... args);
  template<class Service> bool has_service(execution_context& ctx) noexcept;

  template<class T> struct is_executor;

  template<class T>
    constexpr bool is_executor_v = is_executor<T>::value;

  struct executor_arg_t { };
  constexpr executor_arg_t executor_arg = executor_arg_t();
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

```
template<class T, class Executor> struct uses_executor;

template<class T, class Executor>
  constexpr bool uses_executor_v = uses_executor<T, Executor>::value;

template<class T, class Executor = system_executor>
  struct associated_executor;

template<class T, class Executor = system_executor>
  using associated_executor_t = typename associated_executor<T, Executor>::type;

// 13.13, get_associated_executor:

template<class T>
  associated_executor_t<T> get_associated_executor(const T& t) noexcept;
template<class T, class Executor>
  associated_executor_t<T, Executor>
    get_associated_executor(const T& t, const Executor& ex) noexcept;
template<class T, class ExecutionContext>
  associated_executor_t<T, typename ExecutionContext::executor_type>
    get_associated_executor(const T& t, ExecutionContext& ctx) noexcept;

template<class T, class Executor>
  class executor_binder;

template<class T, class Executor, class Signature>
  class async_result<executor_binder<T, Executor>, Signature>;

template<class T, class Executor, class ProtoAllocator>
  struct associated_allocator<executor_binder<T, Executor>, ProtoAllocator>;

template<class T, class Executor, class Executor1>
  struct associated_executor<executor_binder<T, Executor>, Executor1>;

// 13.15, bind_executor:

template<class Executor, class T>
  executor_binder<decay_t<T>, Executor>
    bind_executor(const Executor& ex, T&& t);
template<class ExecutionContext, class T>
  executor_binder<decay_t<T>, typename ExecutionContext::executor_type>
    bind_executor(ExecutionContext& ctx, T&& t);

template<class Executor>
  class executor_work_guard;

// 13.17, make_work_guard:

template<class Executor>
  executor_work_guard<Executor>
    make_work_guard(const Executor& ex);
template<class ExecutionContext>
  executor_work_guard<typename ExecutionContext::executor_type>
    make_work_guard(ExecutionContext& ctx);
template<class T>
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

```
  executor_work_guard<associated_executor_t<T>>
    make_work_guard(const T& t);
template<class T, class U>
  auto make_work_guard(const T& t, U&& u)
    -> decltype(make_work_guard(get_associated_executor(t, forward<U>(u))));

class system_executor;
class system_context;

bool operator==(const system_executor&, const system_executor&);
bool operator!=(const system_executor&, const system_executor&);

class bad_executor;

class executor;

bool operator==(const executor& a, const executor& b) noexcept;
bool operator==(const executor& e, nullptr_t) noexcept;
bool operator==(nullptr_t, const executor& e) noexcept;
bool operator!=(const executor& a, const executor& b) noexcept;
bool operator!=(const executor& e, nullptr_t) noexcept;
bool operator!=(nullptr_t, const executor& e) noexcept;

// 13.22, dispatch:

template<class CompletionToken>
  DEDUCED dispatch(CompletionToken&& token);
template<class Executor, class CompletionToken>
  DEDUCED dispatch(const Executor& ex, CompletionToken&& token);
template<class ExecutionContext, class CompletionToken>
  DEDUCED dispatch(ExecutionContext& ctx, CompletionToken&& token);

// 13.23, post:

template<class CompletionToken>
  DEDUCED post(CompletionToken&& token);
template<class Executor, class CompletionToken>
  DEDUCED post(const Executor& ex, CompletionToken&& token);
template<class ExecutionContext, class CompletionToken>
  DEDUCED post(ExecutionContext& ctx, CompletionToken&& token);

// 13.24, defer:

template<class CompletionToken>
  DEDUCED defer(CompletionToken&& token);
template<class Executor, class CompletionToken>
  DEDUCED defer(const Executor& ex, CompletionToken&& token);
template<class ExecutionContext, class CompletionToken>
  DEDUCED defer(ExecutionContext& ctx, CompletionToken&& token);

template<class Executor>
  class strand;

template<class Executor>
  bool operator==(const strand<Executor>& a, const strand<Executor>& b);
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

```
  template<class Executor>
    bool operator!=(const strand<Executor>& a, const strand<Executor>& b);

  template<class ProtoAllocator = allocator<void>>
    class use_future_t;

  constexpr use_future_t<> use_future = use_future_t<>();

  template<class ProtoAllocator, class Result, class... Args>
    class async_result<use_future_t<ProtoAllocator>, Result(Args...)>;

  template<class Result, class... Args, class Signature>
    class async_result<packaged_task<Result(Args...)>, Signature>;

} // inline namespace v1
} // namespace net
} // namespace experimental

  template<class Allocator>
    struct uses_allocator<experimental::net::v1::executor, Allocator>
      : true_type {};

} // namespace std
```

13.2 Requirements [async.reqmts]

13.2.1 Proto-allocator requirements [async.reqmts.proto.allocator]

1 A type `A` meets the proto-allocator requirements if `A` is `CopyConstructible` (C++ 2014, 17.6.3.1), `Destructible` (C++ 2014, 17.6.3.1), and `allocator_traits<A>::rebind_alloc<U>` meets the allocator requirements (C++ 2014, 17.6.3.5), where `U` is an object type. [*Note:* For example, `allocator<void>` meets the proto-allocator requirements but not the allocator requirements. — *end note*] No comparison operator, copy operation, move operation, or swap operation on these types shall exit via an exception.

13.2.2 Executor requirements [async.reqmts.executor]

1 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.

2 A type `X` meets the `Executor` requirements if it satisfies the requirements of `CopyConstructible` (C++ 2014, 17.6.3.1) and `Destructible` (C++ 2014, 17.6.3.1), as well as the additional requirements listed below.

3 No 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.

4 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. The member function `dispatch` may be recursively reentered.

5 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*]

6 In Table 4, `x1` and `x2` denote (possibly const) values of type `X`, `mx1` denotes an xvalue of type `X`, `f` denotes a function object of `MoveConstructible` (C++ 2014, 17.6.3.1) type `Func` such that `f()` is a valid expression, `a` denotes a (possibly const) value of type `A` where `A` is a type meeting the `ProtoAllocator` requirements

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

(13.2.1), and `u` denotes an identifier.

Table 4 — Executor requirements

expression	type	assertion/note pre/post-conditions
`X u(x1);`		Shall not exit via an exception. *post:* `u == x1` and `std::addressof(u.context()) == std::addressof(x1.context()).`
`X u(mx1);`		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()).`
`x1 == x2`	`bool`	Returns `true` only if `x1` and `x2` can be interchanged with identical effects in any of the expressions defined in these type requirements. [*Note:* Returning `false` does not necessarily imply that the effects are not identical. — *end note*] `operator==` shall be reflexive, symmetric, and transitive, and shall not exit via an exception.
`x1 != x2`	`bool`	Same as `!(x1 == x2).`
`x1.context()`	`execution_context&`, or `E&` where `E` is a type that satisfies the `ExecutionContext` (13.2.3) 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.
`x1.on_work_started()`		Shall not exit via an exception.
`x1.on_work_finished()`		Shall not exit via an exception. *Precondition:* A preceding call `x2.on_work_started()` where `x1 == x2`.
`x1.dispatch(` `  std::move(f), a)`		*Effects:* Creates an object `f1` initialized with *`DECAY_COPY`*`(std::forward<Func>(f))` (C++ 2014, 30.2.6) in the current thread of execution. Calls `f1()` at most once. The executor may block forward progress of the caller until `f1()` finishes execution. 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 document always use the supplied allocator unless otherwise specified. — *end note*] *Synchronization:* The invocation of `dispatch` synchronizes with (C++ 2014, 1.10) the invocation of `f1`.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

Table 4 — Executor requirements (continued)

expression	type	assertion/note pre/post-conditions
`x1.post(std::move(f), a)`		*Effects:* Creates an object `f1` initialized with *`DECAY_COPY`*`(std::forward<Func>(f))` in the current thread of execution. Calls `f1()` at most once. The executor shall not block forward progress of the caller pending completion of `f1()`. The executor may begin `f1`'s progress before the call to `post` completes. 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 document always use the supplied allocator unless otherwise specified. — *end note*] *Synchronization:* The invocation of `post` synchronizes with (C++ 2014, 1.10) the invocation of `f1`.
`x1.defer(std::move(f), a)`		*Effects:* Creates an object `f1` initialized with *`DECAY_COPY`*`(std::forward<Func>(f))` in the current thread of execution. Calls `f1()` at most once. The executor shall not block forward progress of the caller pending completion of `f1()`. The executor should not begin `f1`'s progress before the call to `defer` completes. [*Note:* One use of `defer` is to convey the intention of the caller that `f1` is a continuation of the current call context. The executor can use this information to optimize or otherwise adjust the way in which `f1` is invoked. — *end note*] 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 document always use the supplied allocator unless otherwise specified. — *end note*] *Synchronization:* The invocation of `defer` synchronizes with (C++ 2014, 1.10) the invocation of `f1`.

13.2.3 Execution context requirements [async.reqmts.executioncontext]

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

2 In Table 5, `x` denotes a value of type `X`.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

Table 5 — ExecutionContext requirements

expression	return type	assertion/note pre/post-condition
`X::executor_type`	type meeting `Executor` (13.2.2) requirements	
`x.~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.

13.2.4 Service requirements [async.reqmts.service]

1 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++ 2014, 14.8.2), `is_base_of_v<typename S::key_type, S>` shall be `true`, and `S` shall satisfy the `Destructible` requirements (C++ 2014, 17.6.3.1).

2 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 can be called by the `make_service` function. — *end note*]

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

4 [*Example:*

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

— *end example*]

5 A service's `shutdown` member function shall destroy all copies of function objects that are held by the service.

13.2.5 Signature requirements [async.reqmts.signature]

1 A type satisfies the signature requirements if it is a call signature (C++ 2014, 20.9.1).

13.2.6 Associator requirements [async.reqmts.associator]

1 An associator defines a relationship between different types and objects where, given:

(1.1) — a source object `s` of type `S`,

(1.2) — type requirements `R`, and

(1.3) — a candidate object `c` of type `C` meeting the type requirements `R`,

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

an associated type `A` meeting the type requirements `R` may be computed, and an associated object `a` of type `A` may be obtained.

2 An associator shall be a class template that takes two template type arguments. The first template argument is the source type `S`. The second template argument is the candidate type `C`. The second template argument shall be defaulted to some default candidate type `D` that satisfies the type requirements `R`.

3 An associator shall additionally satisfy the requirements in Table 6. In this table, `X` is a class template that meets the associator requirements, `S` is the source type, `s` is a value of type `S` or `const S`, `C` is the candidate type, `c` is a (possibly const) value of type `C`, `D` is the default candidate type, and `d` is a (possibly const) value of type `D` that is the default candidate object.

Table 6 — Associator requirements

expression	return type	assertion/note pre/post-conditions
`X<S>::type`	`X<S, D>::type`	
`X<S, C>::type`		The associated type.
`X<S>::get(s)`	`X<S>::type`	Returns `X<S>::get(S, d)`.
`X<S, C>::get(s, c)`	`X<S, C>::type`	Returns the associated object.

4 The associator's primary template shall be defined. A program may partially specialize the associator class template for some user-defined type `S`.

5 Finally, the associator shall provide the following type alias and function template in the enclosing namespace:

```
template<class S, class C = D> using X_t = typename X<S, C>::type;

template<class S, class C = D>
typename X<S, C>::type get_X(const S& s, const C& c = d)
{
  return X<S, C>::get(s, c);
}
```

where *X* is replaced with the name of the associator class template. [*Note:* This function template is provided as a convenience, to automatically deduce the source and candidate types. — *end note*]

13.2.7 Requirements on asynchronous operations [async.reqmts.async]

1 This subclause uses the names `Alloc1`, `Alloc2`, `alloc1`, `alloc2`, `Args`, `CompletionHandler`, `completion_handler`, `Executor1`, `Executor2`, `ex1`, `ex2`, `f`, i, N, `Signature`, `token`, $\mathtt{T}_i$, $\mathtt{t}_i$, `work1`, and `work2` as placeholders for specifying the requirements below.

13.2.7.1 General asynchronous operation concepts [async.reqmts.async.concepts]

1 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.

2 The life cycle of an asynchronous operation is comprised of the following events and phases:

(2.1) — Event 1: The asynchronous operation is started by a call to the initiating function.

(2.2) — Phase 1: The asynchronous operation is now outstanding.

(2.3) — Event 2: The externally observable side effects of the asynchronous operation, if any, are fully established. The completion handler is submitted to an executor.

(2.4) — Phase 2: The asynchronous operation is now completed.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

(2.5) — Event 3: The completion handler is called with the result of the asynchronous operation.

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

13.2.7.2 Completion tokens and handlers [async.reqmts.async.token]

1 Initiating functions:

(1.1) — are function templates with template parameter `CompletionToken`;

(1.2) — accept, as the final parameter, a completion token object `token` of type `CompletionToken`;

(1.3) — specify a completion signature, which is a call signature (C++ 2014, 20.9.1) `Signature` that determines the arguments to the completion handler.

2 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 `std::forward<CompletionToken>(token)`. [*Note:* No other requirements are placed on the type `CompletionToken`. — *end note*]

3 The type `CompletionHandler` shall satisfy the requirements of `Destructible` (C++ 2014, 17.6.3.1) and `MoveConstructible` (C++ 2014, 17.6.3.1), and be callable with the specified call signature.

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

13.2.7.3 Deduction of initiating function return type [async.reqmts.async.return.type]

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

2 For the sake of exposition, this document 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`.

13.2.7.4 Production of initiating function return value [async.reqmts.async.return.value]

1 An initiating function produces its return type as follows:

(1.1) — constructing an object `result` of type `async_result<decay_t<CompletionToken>, Signature>`, initialized as `result(completion_handler)`; and

(1.2) — using `result.get()` as the operand of the return statement.

2 [*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();
}
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

3 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*]

13.2.7.5 Lifetime of initiating function arguments [async.reqmts.async.lifetime]

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

(1.1) — 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 is responsible for guaranteeing the validity of the argument until the completion handler is invoked. — *end note*]

(1.2) — Otherwise, the implementation does 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.

13.2.7.6 Non-blocking requirements on initiating functions[async.reqmts.async.non.blocking]

1 An initiating function shall not block (C++ 2014, 17.3.2) the calling thread pending completion of the outstanding operation.

2 [*Note:* Initiating functions can still block the calling thread for other reasons. For example, if an initiating function locks a mutex in order to synchronize access to shared data. — *end note*]

13.2.7.7 Associated executor [async.reqmts.async.assoc.exec]

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

13.2.7.8 I/O executor [async.reqmts.async.io.exec]

1 An asynchronous operation has an associated executor satisfying the `Executor` (13.2.2) requirements. If not otherwise specified by the asynchronous operation, this associated executor is an object of type `system_executor`.

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

(2.1) — If the initiating function is a member function, the associated executor is that returned by the `get_executor` member function on the same object.

(2.2) — 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.

3 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.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

13.2.7.9 Completion handler executor [async.reqmts.async.handler.exec]

1 A completion handler object of type `CompletionHandler` has an associated executor satisfying the Executor requirements (13.2.2). The type of this associated executor is `associated_executor_t<CompletionHandler, Executor1>`. Let `Executor2` be the type `associated_executor_t<CompletionHandler, Executor1>`. Let `ex2` be a value of type `Executor2` obtained by performing `get_associated_executor(completion_handler, ex1)`.

13.2.7.10 Outstanding work [async.reqmts.async.work]

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

(1.1) — an object `work1` of type `executor_work_guard<Executor1>`, initialized as `work1(ex1)`, and where `work1.owns_work() == true`; and

(1.2) — an object `work2` of type `executor_work_guard<Executor2>`, initialized as `work2(ex2)`, and where `work2.owns_work() == true`.

13.2.7.11 Allocation of intermediate storage [async.reqmts.async.alloc]

1 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*]

2 Let `Alloc1` be a type, satisfying the `ProtoAllocator` (13.2.1) requirements, that represents the asynchronous operation's default allocation strategy. [*Note:* Typically `allocator<void>`. — *end note*] Let `alloc1` be a value of type `Alloc1`.

3 A completion handler object of type `CompletionHandler` has an associated allocator object `alloc2` of type `Alloc2` satisfying the `ProtoAllocator` (13.2.1) 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)`.

4 The asynchronous operations defined in this document:

(4.1) — If required, allocate memory using only the completion handler's associated allocator.

(4.2) — Prior to completion handler execution, deallocate any memory allocated using the completion handler's associated allocator.

5 [*Note:* The implementation can perform operating system or underlying API calls that perform memory allocations not using the associated allocator. Invocations of the allocator functions do not introduce data races (see C++ 2014, 17.6.5.9). — *end note*]

13.2.7.12 Execution of completion handler on completion of asynchronous operation [async.reqmts.async.completion]

1 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 $\mathtt{T}_i$ be the ith type in `Args...` and let $\mathtt{t}_i$ be the ith completion handler argument associated with $\mathtt{T}_i$.

2 Let `f` be a function object, callable as `f()`, that invokes `completion_handler` as if by `completion_handler(forward<T_0>(t_0), ..., forward<T_{N-1}>(t_{N-1}))`.

3 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)`.

13.2.7.13 Completion handlers and exceptions [async.reqmts.async.exceptions]

1 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.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

13.2.7.14 Composed asynchronous operations [async.reqmts.async.composed]

1 In this document, a *composed asynchronous operation* is an asynchronous operation that is implemented in terms of zero or more intermediate calls to other asynchronous operations. The intermediate asynchronous operations are performed sequentially. [*Note:* That is, the completion handler of an intermediate operation initiates the next operation in the sequence. — *end note*]

An intermediate operation's completion handler shall have an associated executor that is either:

(1.1) — the type `Executor2` and object `ex2` obtained from the completion handler type `CompletionHandler` and object `completion_handler`; or

(1.2) — an object of an unspecified type satisfying the Executor requirements (13.2.2), that delegates executor operations to the type `Executor2` and object `ex2`.

An intermediate operation's completion handler shall have an associated allocator that is either:

(1.3) — the type `Alloc2` and object `alloc2` obtained from the completion handler type `CompletionHandler` and object `completion_handler`; or

(1.4) — an object of an unspecified type satisfying the ProtoAllocator requirements (13.2.1), that delegates allocator operations to the type `Alloc2` and object `alloc2`.

13.3 Class template `async_result` [async.async.result]

1 The `async_result` class template is a customization point for asynchronous operations. Template parameter `CompletionToken` specifies the model used to obtain the result of the asynchronous operation. Template parameter `Signature` is the call signature (C++ 2014, 20.9.1) for the completion handler type invoked on completion of the asynchronous operation. The `async_result` template:

(1.1) — transforms a `CompletionToken` into a completion handler type that is based on a `Signature`; and

(1.2) — determines the return type and return value of an asynchronous operation's initiating function.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class CompletionToken, class Signature>
  class async_result
  {
  public:
    using completion_handler_type = CompletionToken;
    using return_type = void;

    explicit async_result(completion_handler_type&) {}
    async_result(const async_result&) = delete;
    async_result& operator=(const async_result&) = delete;

    return_type get() {}
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

2 The template parameter `CompletionToken` shall be an object type. The template parameter `Signature` shall be a call signature (C++ 2014, 20.9.1).

3 Specializations of `async_result` shall satisfy the `Destructible` requirements (C++ 2014, 17.6.3.1) in addition to the requirements in Table 7. In this table, `R` is a specialization of `async_result`; `r` is a modifiable lvalue of type `R`; and `h` is a modifiable lvalue of type `R::completion_handler_type`.

Table 7 — `async_result` specialization requirements

Expression	Return type	Requirement
`R::completion_handler_type`		A type satisfying `MoveConstructible` requirements (C++ 2014, 17.6.3.1), An object of type `completion_handler_type` shall be a function object with call signature `Signature`, and `completion_handler_type` shall be constructible with an rvalue of type `CompletionToken`.
`R::return_type`		`void`; or a type satisfying `MoveConstructible` requirements (C++ 2014, 17.6.3.1)
`R r(h);`		
`r.get()`	`R::return_type`	[*Note:* An asynchronous operation's initiating function uses the `get()` member function as the sole operand of a return statement. — *end note*]

13.4 Class template `async_completion` [async.async.completion]

1 Class template `async_completion` is provided as a convenience, to simplify the implementation of asynchronous operations that use `async_result`.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class CompletionToken, class Signature>
  struct async_completion
  {
    using completion_handler_type = async_result<decay_t<CompletionToken>,
      Signature>::completion_handler_type;

    explicit async_completion(CompletionToken& t);
    async_completion(const async_completion&) = delete;
    async_completion& operator=(const async_completion&) = delete;

    see below completion_handler;
    async_result<decay_t<CompletionToken>, Signature> result;
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

2 The template parameter `Signature` shall be a call signature (C++ 2014, 20.9.1).

```
explicit async_completion(CompletionToken& t);
```

3 *Effects:* If `CompletionToken` and `completion_handler_type` are the same type, binds `completion_handler` to `t`; otherwise, initializes `completion_handler` with `std::forward<CompletionToken>(t)`. Initializes `result` with `completion_handler`.

```
see below completion_handler;
```

4 *Type:* `completion_handler_type&` if `CompletionToken` and `completion_handler_type` are the same type; otherwise, `completion_handler_type`.

13.5 Class template `associated_allocator` [async.assoc.alloc]

1 Class template `associated_allocator` is an associator (13.2.6) for the `ProtoAllocator` (13.2.1) type requirements, with default candidate type `allocator<void>` and default candidate object `allocator<void>()`.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class ProtoAllocator = allocator<void>>
  struct associated_allocator
  {
    using type = see below;

    static type get(const T& t, const ProtoAllocator& a = ProtoAllocator()) noexcept;
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Specializations of `associated_allocator` shall satisfy the requirements in Table 8. In this table, `X` is a specialization of `associated_allocator` for the template parameters `T` and `ProtoAllocator`; `t` is a (possibly const) value of type `T`; and `a` is an object of type `ProtoAllocator`.

Table 8 — `associated_allocator` specialization requirements

Expression	Return type	Note
`typename X::type`	A type meeting the proto-allocator (13.2.1) requirements.	
`X::get(t)`	`X::type`	Shall not exit via an exception. Equivalent to `X::get(t, ProtoAllocator())`.
`X::get(t, a)`	`X::type`	Shall not exit via an exception.

13.5.1 `associated_allocator` members [async.assoc.alloc.members]

```
using type = see below;
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TS 19216:2018

1 *Type:* `T::allocator_type` if the *qualified-id* `T::allocator_type` is valid and denotes a type (C++ 2014, 14.8.2). Otherwise `ProtoAllocator`.

```
type get(const T& t, const ProtoAllocator& a = ProtoAllocator()) noexcept;
```

2 *Returns:* `t.get_allocator()` if the *qualified-id* `T::allocator_type` is valid and denotes a type (C++ 2014, 14.8.2). Otherwise `a`.

13.6 Function `get_associated_allocator` [async.assoc.alloc.get]

```
template<class T>
  associated_allocator_t<T> get_associated_allocator(const T& t) noexcept;
```

1 *Returns:* `associated_allocator<T>::get(t)`.

```
template<class T, class ProtoAllocator>
  associated_allocator_t<T, ProtoAllocator>
    get_associated_allocator(const T& t, const ProtoAllocator& a) noexcept;
```

2 *Returns:* `associated_allocator<T, ProtoAllocator>::get(t, a)`.

13.7 Class `execution_context` [async.exec.ctx]

1 Class `execution_context` implements an extensible, type-safe, polymorphic set of services, indexed by service type.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class execution_context
  {
  public:
    class service;

    // 13.7.1, construct / copy / destroy:

    execution_context();
    execution_context(const execution_context&) = delete;
    execution_context& operator=(const execution_context&) = delete;
    virtual ~execution_context();

    // 13.7.3, execution context operations:

    void notify_fork(fork_event e);

  protected:

    // 13.7.4, execution context protected operations:

    void shutdown() noexcept;
    void destroy() noexcept;
  };

  // service access:
  template<class Service> typename Service::key_type&
    use_service(execution_context& ctx);
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

```
  template<class Service, class... Args> Service&
    make_service(execution_context& ctx, Args&&... args);
  template<class Service> bool has_service(const execution_context& ctx) noexcept;
  class service_already_exists : public logic_error { };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Access to the services of an `execution_context` is via three function templates, `use_service`, `make_service` and `has_service`.

3 In a call to `use_service<Service>`, the type argument chooses a service, identified by `Service::key_type`, from a set of services in an `execution_context`. If the service is not present in the `execution_context`, an object of type `Service` is created and added to the `execution_context`. A program can check if an `execution_context` implements a particular service with the function template `has_service<Service>`.

4 Service objects may be explicitly added to an `execution_context` using the function template `make_service<Service>`. If the service is already present, `make_service` exits via an exception of type `service_already_exists`.

5 Once a service reference is obtained from an `execution_context` object by calling `use_service`, that reference remains usable until a call to `destroy()`.

6 If a call to a specialization of `use_service` or `make_service` recursively calls another specialization of `use_service` or `make_service` which would choose the same service (identified by `key_type`) from the same `execution_context`, then the behavior is undefined. [*Note:* Nested calls to specializations for different service types are well-defined. — *end note*]

13.7.1 `execution_context` constructor [async.exec.ctx.cons]

`execution_context();`

1 *Effects:* Creates an object of class `execution_context` which contains no services. [*Note:* An implementation can preload services of internal service types for its own use. — *end note*]

13.7.2 `execution_context` destructor [async.exec.ctx.dtor]

`~execution_context();`

1 *Effects:* Destroys an object of class `execution_context`. Performs `shutdown()` followed by `destroy()`.

13.7.3 `execution_context` operations [async.exec.ctx.ops]

`void notify_fork(fork_event e);`

1 *Effects:* For each service object `svc` in the set:

(1.1) — If `e == fork_event::prepare`, performs `svc->notify_fork(e)` in reverse order of addition to the set.

(1.2) — Otherwise, performs `svc->notify_fork(e)` in order of addition to the set.

13.7.4 `execution_context` protected operations [async.exec.ctx.protected]

`void shutdown() noexcept;`

1 *Effects:* For each service object `svc` in the `execution_context` set, in reverse order of addition to the set, performs `svc->shutdown()`. For each service in the set, `svc->shutdown()` is called only once irrespective of the number of calls to `shutdown` on the `execution_context`.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

```
void destroy() noexcept;
```

2 *Effects:* Destroys each service object in the `execution_context` set, and removes it from the set, in reverse order of addition to the set.

13.7.5 `execution_context` globals [async.exec.ctx.globals]

1 The functions `use_service`, `make_service`, and `has_service` do not introduce data races as a result of concurrent calls to those functions from different threads.

```
template<class Service> typename Service::key_type&
  use_service(execution_context& ctx);
```

2 *Effects:* If an object of type `Service::key_type` does not already exist in the `execution_context` set identified by `ctx`, creates an object of type `Service`, initialized as `Service(ctx)`, and adds it to the set.

3 *Returns:* A reference to the corresponding service of `ctx`.

4 *Remarks:* The reference returned remains valid until a call to `destroy`.

```
template<class Service, class... Args> Service&
  make_service(execution_context& ctx, Args&&... args);
```

5 *Requires:* A service object of type `Service::key_type` does not already exist in the `execution_context` set identified by `ctx`.

6 *Effects:* Creates an object of type `Service`, initialized as `Service(ctx, forward<Args>(args)...)`, and adds it to the `execution_context` set identified by `ctx`.

7 *Throws:* `service_already_exists` if a corresponding service object of type `Service::key_type` is already present in the set.

8 *Remarks:* The reference returned remains valid until a call to `destroy`.

```
template<class Service> bool has_service(const execution_context& ctx) noexcept;
```

9 *Returns:* `true` if an object of type `Service::key_type` is present in `ctx`, otherwise `false`.

13.8 Class `execution_context::service` [async.exec.ctx.svc]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class execution_context::service
  {
  protected:
    // construct / copy / destroy:

    explicit service(execution_context& owner);
    service(const service&) = delete;
    service& operator=(const service&) = delete;
    virtual ~service();

    // service observers:

    execution_context& context() noexcept;
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TS 19216:2018

```
    private:
      // service operations:

      virtual void shutdown() noexcept = 0;
      virtual void notify_fork(fork_event e) {}

      execution_context& context_; // exposition only
    };

  } // inline namespace v1
  } // namespace net
  } // namespace experimental
  } // namespace std
```

`explicit service(execution_context& owner);`

1 *Postconditions:* `std::addressof(context_) == std::addressof(owner)`.

`execution_context& context() noexcept;`

2 *Returns:* `context_`.

13.9 Class template `is_executor` [async.is.exec]

1 The class template `is_executor` can be used to detect executor types satisfying the `Executor` (13.2.2) type requirements.

```
  namespace std {
  namespace experimental {
  namespace net {
  inline namespace v1 {

    template<class T> struct is_executor;

  } // inline namespace v1
  } // namespace net
  } // namespace experimental
  } // namespace std
```

2 `T` shall be a complete type.

3 Class template `is_executor` is a UnaryTypeTrait (C++ 2014, 20.10.1) with a BaseCharacteristic of `true_type` if the type `T` meets the syntactic requirements for `Executor` (13.2.2), otherwise `false_type`.

13.10 Executor argument tag [async.executor.arg]

```
  namespace std {
  namespace experimental {
  namespace net {
  inline namespace v1 {

    struct executor_arg_t { };
    constexpr executor_arg_t executor_arg = executor_arg_t();

  } // inline namespace v1
  } // namespace net
  } // namespace experimental
  } // namespace std
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

1 The `executor_arg_t` struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, types may have constructors with `executor_arg_t` as the first argument, immediately followed by an argument of a type that satisfies the Executor requirements (13.2.2).

13.11 `uses_executor` [async.uses.executor]

13.11.1 `uses_executor` trait [async.uses.executor.trait]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor> struct uses_executor;

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 Remark: Detects whether `T` has a nested `executor_type` that is convertible from `Executor`. Meets the `BinaryTypeTrait` requirements (C++ 2014, 20.10.1). The implementation provides a definition that is derived from `true_type` if a type `T::executor_type` exists and `is_convertible<Executor, T::executor_type>::value != false`, otherwise it is derived from `false_type`. A program may specialize this template to derive from `true_type` for a user-defined type `T` that does not have a nested `executor_type` but nonetheless can be constructed with an executor if the first argument of a constructor has type `executor_arg_t` and the second argument has type `Executor`.

13.11.2 uses-executor construction [async.uses.executor.cons]

1 Uses-executor construction with executor `Executor` refers to the construction of an object `obj` of type `T`, using constructor arguments `v1, v2, ..., vN` of types `V1, V2, ..., VN`, respectively, and an executor `ex` of type `Executor`, according to the following rules:

(1.1) — if `uses_executor_v<T, Executor>` is `true` and `is_constructible<T, executor_arg_t, Executor, V1, V2, ..., VN>::value` is `true`, then `obj` is initialized as `obj(executor_arg, ex, v1, v2, ..., vN)`;

(1.2) — otherwise, `obj` is initialized as `obj(v1, v2, ..., vN)`.

13.12 Class template `associated_executor` [async.assoc.exec]

1 Class template `associated_executor` is an associator (13.2.6) for the `Executor` (13.2.2) type requirements, with default candidate type `system_executor` and default candidate object `system_executor()`.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor = system_executor>
  struct associated_executor
  {
    using type = see below;

    static type get(const T& t, const Executor& e = Executor()) noexcept;
  };
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

```
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Specializations of `associated_executor` shall satisfy the requirements in Table 9. In this table, `X` is a specialization of `associated_executor` for the template parameters `T` and `Executor`; `t` is a (possible const) value of `T`; and `e` is an object of type `Executor`.

Table 9 — `associated_executor` specialization requirements

Expression	Return type	Note
`typename X::type`	A type meeting Executor requirements (13.2.2).	
`X::get(t)`	`X::type`	Shall not exit via an exception. Equivalent to `X::get(t, Executor())`.
`X::get(t, e)`	`X::type`	Shall not exit via an exception.

13.12.1 `associated_executor` members [async.assoc.exec.members]

```
using type = see below;
```

1 *Type:* `T::executor_type` if the *qualified-id* `T::executor_type` is valid and denotes a type (C++ 2014, 14.8.2). Otherwise `Executor`.

```
type get(const T& t, const Executor& e = Executor()) noexcept;
```

2 *Returns:* `t.get_executor()` if the *qualified-id* `T::executor_type` is valid and denotes a type (C++ 2014, 14.8.2). Otherwise `e`.

13.13 Function `get_associated_executor` [async.assoc.exec.get]

```
template<class T>
  associated_executor_t<T> get_associated_executor(const T& t) noexcept;
```

1 *Returns:* `associated_executor<T>::get(t)`.

```
template<class T, class Executor>
  associated_executor_t<T, Executor>
    get_associated_executor(const T& t, const Executor& ex) noexcept;
```

2 *Returns:* `associated_executor<T, Executor>::get(t, ex)`.

3 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<Executor>` is `true`.

```
template<class T, class ExecutionContext>
  associated_executor_t<T, typename ExecutionContext::executor_type>
    get_associated_executor(const T& t, ExecutionContext& ctx) noexcept;
```

4 *Returns:* `get_associated_executor(t, ctx.get_executor())`.

5 *Remarks:* This function shall not participate in overload resolution unless `is_convertible<ExecutionContext&, execution_context&>::value` is `true`.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TS 19216:2018

13.14 Class template executor_binder [async.exec.binder]

1 The class template executor_binder binds executors to objects. A specialization executor_binder<T, Executor> binds an executor of type Executor satisfying the Executor requirements (13.2.2) to an object or function of type T.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor>
  class executor_binder
  {
  public:
    // types:

    using target_type = T;
    using executor_type = Executor;

    // 13.14.1, construct / copy / destroy:

    executor_binder(T t, const Executor& ex);
    executor_binder(const executor_binder& other) = default;
    executor_binder(executor_binder&& other) = default;
    template<class U, class OtherExecutor>
      executor_binder(const executor_binder<U, OtherExecutor>& other);
    template<class U, class OtherExecutor>
      executor_binder(executor_binder<U, OtherExecutor>&& other);
    template<class U, class OtherExecutor>
      executor_binder(executor_arg_t, const Executor& ex,
        const executor_binder<U, OtherExecutor>& other);
    template<class U, class OtherExecutor>
      executor_binder(executor_arg_t, const Executor& ex,
        executor_binder<U, OtherExecutor>&& other);

    ~executor_binder();

    // 13.14.2, executor binder access:

    T& get() noexcept;
    const T& get() const noexcept;
    executor_type get_executor() const noexcept;

    // 13.14.3, executor binder invocation:

    template<class... Args>
      result_of_t<T&(Args&&...)> operator()(Args&&... args);
    template<class... Args>
      result_of_t<const T&(Args&&...)> operator()(Args&&... args) const;

  private:
    Executor ex_; // exposition only
    T target_; // exposition only
  };
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

```
    template<class T, class Executor, class Signature>
      class async_result<executor_binder<T, Executor>, Signature>;

    template<class T, class Executor, class ProtoAllocator>
      struct associated_allocator<executor_binder<T, Executor>, ProtoAllocator>;

    template<class T, class Executor, class Executor1>
      struct associated_executor<executor_binder<T, Executor>, Executor1>;

  } // inline namespace v1
  } // namespace net
  } // namespace experimental
  } // namespace std
```

13.14.1 `executor_binder` constructors [async.exec.binder.cons]

```
executor_binder(T t, const Executor& ex);
```

1 *Effects:* Initializes `ex_` with `ex`. Initializes `target_` by performing uses-executor construction, using the constructor argument `std::move(t)` and the executor `ex_`.

```
template<class U, class OtherExecutor>
  executor_binder(const executor_binder<U, OtherExecutor>& other);
```

2 *Requires:* If `U` is not convertible to `T`, or if `OtherExecutor` is not convertible to `Executor`, the program is ill-formed.

3 *Effects:* Initializes `ex_` with `other.get_executor()`. Initializes `target_` by performing uses-executor construction, using the constructor argument `other.get()` and the executor `ex_`.

```
template<class U, class OtherExecutor>
  executor_binder(executor_binder<U, OtherExecutor>&& other);
```

4 *Requires:* If `U` is not convertible to `T`, or if `OtherExecutor` is not convertible to `Executor`, the program is ill-formed.

5 *Effects:* Initializes `ex_` with `other.get_executor()`. Initializes `target_` by performing uses-executor construction, using the constructor argument `std::move(other.get())` and the executor `ex_`.

```
template<class U, class OtherExecutor>
  executor_binder(executor_arg_t, const Executor& ex,
    const executor_binder<U, OtherExecutor>& other);
```

6 *Requires:* If `U` is not convertible to `T` the program is ill-formed.

7 *Effects:* Initializes `ex_` with `ex`. Initializes `target_` by performing uses-executor construction, using the constructor argument `other.get()` and the executor `ex_`.

```
template<class U, class OtherExecutor>
  executor_binder(executor_arg_t, const Executor& ex,
    executor_binder<U, OtherExecutor>&& other);
```

8 *Requires:* `U` is `T` or convertible to `T`.

9 *Effects:* Initializes `ex_` with `ex`. Initializes `target_` by performing uses-executor construction, using the constructor argument `std::move(other.get())` and the executor `ex_`.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

13.14.2 executor_binder access [async.exec.binder.access]

```
T& get() noexcept;
const T& get() const noexcept;
```

1 *Returns:* `target_`.

```
executor_type get_executor() const noexcept;
```

2 *Returns:* `executor_`.

13.14.3 executor_binder invocation [async.exec.binder.invocation]

```
template<class... Args>
  result_of_t<T&(Args&&...)> operator()(Args&&... args);
template<class... Args>
  result_of_t<const T&(Args&&...)> operator()(Args&&... args) const;
```

1 *Returns:* *INVOKE*`(get(), forward<Args>(args)...)` (C++ 2014, 20.9.2).

13.14.4 Class template partial specialization async_result [async.exec.binder.async.result]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor, class Signature>
  class async_result<executor_binder<T, Executor>, Signature>
  {
  public:
    using completion_handler_type = executor_binder<
      typename async_result<T, Signature>::completion_handler_type,
        Executor>;
    using return_type = typename async_result<T, Signature>::return_type;

    explicit async_result(completion_handler_type& h);
    async_result(const async_result&) = delete;
    async_result& operator=(const async_result&) = delete;

    return_type get();

  private:
    async_result<T, Signature> target_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

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

1 *Effects:* Initializes `target_` as `target_(h.get())`.

```
return_type get();
```

2 *Returns:* `target_.get()`.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

13.14.5 Class template partial specialization `associated_allocator` [async.exec.binder.assoc.alloc]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor, class ProtoAllocator>
    struct associated_allocator<executor_binder<T, Executor>, ProtoAllocator>
  {
    using type = associated_allocator_t<T, ProtoAllocator>;

    static type get(const executor_binder<T, Executor>& b,
                    const ProtoAllocator& a = ProtoAllocator()) noexcept;
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

```
static type get(const executor_binder<T, Executor>& b,
                const ProtoAllocator& a = ProtoAllocator()) noexcept;
```

1 *Returns:* `associated_allocator<T, ProtoAllocator>::get(b.get(), a)`.

13.14.6 Class template partial specialization `associated_executor` [async.exec.binder.assoc.exec]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor, class Executor1>
    struct associated_executor<executor_binder<T, Executor>, Executor1>
  {
    using type = Executor;

    static type get(const executor_binder<T, Executor>& b,
                    const Executor1& e = Executor1()) noexcept;
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

```
static type get(const executor_binder<T, Executor>& b,
                const Executor1& e = Executor1()) noexcept;
```

1 *Returns:* `b.get_executor()`.

13.15 Function `bind_executor` [async.bind.executor]

```
template<class Executor, class T>
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

```
executor_binder<decay_t<T>, Executor>
  bind_executor(const Executor& ex, T&& t);
```

1 *Returns:* `executor_binder<decay_t<T>, Executor>(forward<T>(t), ex)`.

2 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<Executor>` is `true`.

```
template<class ExecutionContext, class CompletionToken>
  executor_binder<decay_t<T>, typename ExecutionContext::executor_type>
    bind_executor(ExecutionContext& ctx, T&& t);
```

3 *Returns:* `bind_executor(ctx.get_executor(), forward<T>(t))`.

4 *Remarks:* This function shall not participate in overload resolution unless `is_convertible<ExecutionContext&, execution_context&>::value` is `true`.

13.16 Class template `executor_work_guard` [async.exec.work.guard]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Executor>
  class executor_work_guard
  {
  public:
    // types:

    using executor_type = Executor;

    // construct / copy / destroy:

    explicit executor_work_guard(const executor_type& ex) noexcept;
    executor_work_guard(const executor_work_guard& other) noexcept;
    executor_work_guard(executor_work_guard&& other) noexcept;

    executor_work_guard& operator=(const executor_work_guard&) = delete;

    ~executor_work_guard();

    // executor work guard observers:

    executor_type get_executor() const noexcept;
    bool owns_work() const noexcept;

    // executor work guard modifiers:

    void reset() noexcept;

  private:
    Executor ex_; // exposition only
    bool owns_; // exposition only
  };

} // inline namespace v1
} // namespace net
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TS 19216:2018

```
} // namespace experimental
} // namespace std
```

13.16.1 executor_work_guard members [async.exec.work.guard.members]

```
explicit executor_work_guard(const executor_type& ex) noexcept;
```

1 *Effects:* Initializes `ex_` with `ex`, and then performs `ex_.on_work_started()`.

2 *Postconditions:* `ex_ == ex` and `owns_ == true`.

```
executor_work_guard(const executor_work_guard& other) noexcept;
```

3 *Effects:* Initializes `ex_` with `other.ex_`. If `other.owns_ == true`, performs `ex_.on_work_started()`.

4 *Postconditions:* `ex_ == other.ex_` and `owns_ == other.owns_`.

```
executor_work_guard(executor_work_guard&& other) noexcept;
```

5 *Effects:* Initializes `ex_` with `std::move(other.ex_)` and initializes `owns_` with `other.owns_`, and sets `other.owns_` to `false`.

```
~executor_work_guard();
```

6 *Effects:* If `owns_` is `true`, performs `ex_.on_work_finished()`.

```
executor_type get_executor() const noexcept;
```

7 *Returns:* `ex_`.

```
bool owns_work() const noexcept;
```

8 *Returns:* `owns_`.

```
void reset() noexcept;
```

9 *Effects:* If `owns_` is `true`, performs `ex_.on_work_finished()`.

10 *Postconditions:* `owns_ == false`.

13.17 Function make_work_guard [async.make.work.guard]

```
template<class Executor>
  executor_work_guard<Executor>
    make_work_guard(const Executor& ex);
```

1 *Returns:* `executor_work_guard<Executor>(ex)`.

2 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<Executor>` is `true`.

```
template<class ExecutionContext>
  executor_work_guard<typename ExecutionContext::executor_type>
    make_work_guard(ExecutionContext& ctx);
```

3 *Returns:* `make_work_guard(ctx.get_executor())`.

4 *Remarks:* This function shall not participate in overload resolution unless `is_convertible<ExecutionContext&, execution_context&>::value` is `true`.

```
template<class T>
  executor_work_guard<associated_executor_t<T>>
    make_work_guard(const T& t);
```

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018

5 *Returns:* `make_work_guard(get_associated_executor(t))`.

6 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<T>` is `false` and `is_convertible<T&, execution_context&>::value` is `false`.

```
template<class T, class U>
  auto make_work_guard(const T& t, U&& u)
    -> decltype(make_work_guard(get_associated_executor(t, forward<U>(u))));
```

7 *Returns:* `make_work_guard(get_associated_executor(t, forward<U>(u)))`.

13.18 Class `system_executor` [async.system.exec]

1 Class `system_executor` represents a set of rules where function objects are permitted to execute on any thread.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class system_executor
  {
  public:
    // constructors:

    system_executor() {}

    // 13.18.1, executor operations:

    system_context& context() const noexcept;

    void on_work_started() const noexcept {}
    void on_work_finished() const noexcept {}

    template<class Func, class ProtoAllocator>
      void dispatch(Func&& f, const ProtoAllocator& a) const;
    template<class Func, class ProtoAllocator>
      void post(Func&& f, const ProtoAllocator& a) const;
    template<class Func, class ProtoAllocator>
      void defer(Func&& f, const ProtoAllocator& a) const;
  };

  bool operator==(const system_executor&, const system_executor&) noexcept;
  bool operator!=(const system_executor&, const system_executor&) noexcept;

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Class `system_executor` satisfies the `Destructible` (C++ 2014, 17.6.3.1), `DefaultConstructible` (C++ 2014, 17.6.3.1), and `Executor` (13.2.2) type requirements.

3 To satisfy the `Executor` requirements for the `post` and `defer` member functions, the system executor may create `thread` objects to run the submitted function objects. These `thread` objects are collectively referred to as system threads.

 © ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TS 19216:2018