BS ISO/IEC 19516:2020

BSI Standards Publication

Information technology — Object management group — Interface definition language (IDL) 4.2

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 19516:2020

National foreword

This British Standard is the UK implementation of ISO/IEC 19516:2020.

The UK participation in its preparation was entrusted to Technical Committee ICT/1, Information systems co-ordination.

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 2020
Published by BSI Standards Limited 2020

ISBN 978 0 539 02468 5

ICS 35.060

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

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 29 February 2020.

Amendments/corrigenda issued since publication

Date	Text affected

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 19516:2020

INTERNATIONAL STANDARD

ISO/IEC 19516

First edition
2020-02

Information technology — Object management group — Interface definition language (IDL) 4.2

Reference number
ISO/IEC 19516:2020(E)

Documents
Document Sharing Hub

© ISO/IEC 2020

Full document details: BS ISO/IEC 19516:2020

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2020

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 2020 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 19516:2020

Contents

Foreword v

Introduction v

1 Scope 1
1.1 Overview 1

2 Conformance Criteria 1

3 Normative References 2

4 Terms and Definitions 2

5 Symbols 2

6 Additional Information 3
6.1 Acknowledgments 3
6.2 History 3

7 IDL Syntax and Semantics 4
7.1 Overview 4
7.2 Lexical Conventions 5
7.2.1 Tokens 8
7.2.2 Comments 8
7.2.3 Identifiers 8
7.2.4 Keywords 9
7.2.5 Other Characters Recognized by IDL 10
7.2.6 Literals 11
7.3 Preprocessing 13
7.4 IDL Grammar 13
7.4.1 Building Block Core Data Types 14
7.4.2 Building Block Any 30
7.4.3 Building Block Interfaces — Basic 31
7.4.4 Building Block Interfaces — Full 37
7.4.5 Building Block Value Types 39
7.4.6 Building Block CORBA-Specific — Interfaces 42
7.4.7 Building Block CORBA-Specific — Value Types 48
7.4.8 Building Block Components — Basic 53
7.4.9 Building Block Components — Homes 56
7.4.10 Building Block CCM-Specific 59
7.4.11 Building Block Components — Ports and Connectors 64
7.4.12 Building Block Template Modules 67
7.4.13 Building Block Extended Data-Types 70
7.4.14 Building Block Anonymous Types 76
7.4.15 Building Block Annotations 77
7.4.16 Relationships between the Building Blocks 80
7.5 Names and Scoping 81
7.5.1 Qualified Names 81
7.5.2 Scoping Rules and Name Resolution 82
7.5.3 Special Scoping Rules for Type Names 84

8 Standardized Annotations 86
8.1 Overview 86
8.2 Introduction 86
8.2.1 Rules for Defining Standardized Annotations 86
8.2.2 Rules for Using Standardized Annotations 86
8.3 Standardized Groups of Annotations 86
8.3.1 Group of Annotations General Purpose 86
8.3.2 Group of Annotations Data Modeling 89

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 19516:2020

8.3.3 Group of Annotations: Units and Ranges 89
8.3.4 Group of Annotations Data Implementation 91
8.3.5 Group of Annotations Code Generation 91
8.3.6 Group of Annotations Interfaces 92

9 Profiles 93
9.1 Overview 93
9.2 CORBA and CCM Profiles 93
9.2.1 Plain CORBA Profile 93
9.2.2 Minimum CORBA Profile 94
9.2.3 CCM Profile 94
9.2.4 CCM with Generic Interaction Support Profile 94
9.3 DDS Profiles 95
9.3.1 Plain DDS Profile 95
9.3.2 Extensible DDS Profile 95
9.3.3 RPC over DDS Profile 95

Annex A Consolidated IDL Grammar 97

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

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.

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 document should be noted (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 and IEC 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) or the IEC list of patent declarations received (see http://patents.iec.ch).

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

For an explanation of 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 www.iso.org/iso/foreword.html.

This document was prepared by the Object Management Group (OMG) (as the OMG specification for Interface Definition Language (IDL), v4.2) and drafted in accordance with its editorial rules. It was adopted, under the JTC 1 PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

This document is related to:

- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1995, Information Technology — Open Distributed Processing — Reference Model: Foundations
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1995, Information Technology — Open Distributed Processing — Reference Model: Architecture
- ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1997, Information Technology — Open Distributed Processing — Interface Definition Language

Apart from this Foreword, the text of this document is identical with that for the OMG specification for Interface Definition Language (IDL), v4.2.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

Introduction

The rapid growth of distributed processing has led to a need for a coordinating framework for this standardization and ITU-T Recommendations X.901-904 | ISO/IEC 10746, the Reference Model of Open Distributed Processing (RM-ODP) provides such a framework. It defines an architecture within which support of distribution, interoperability, and portability can be integrated.

RM-ODP Part 2 (ISO RM-ODP Part 2 (ISO/IEC 10746-2) defines the foundational concepts and modeling framework for describing distributed systems. The scopes and objectives of the RM-ODP Part 2 and the UML, while related, are not the same and, in a number of cases, the RM-ODP Part 2 and the UML specification use the same term for concepts which are related but not identical (e.g., interface). Nevertheless, a specification using the Part 2 modeling concepts can be expressed using UML with appropriate extensions (using stereotypes, tags, and constraints).

RM-ODP Part 3 (ISO/IEC 10746-3) specifies a generic architecture of open distributed systems, expressed using the foundational concepts and framework defined in Part 2. Given the relation between UML as a modeling language and Part 3 of the RM-ODP standard, it is easy to show that UML is suitable as a notation for the individual viewpoint specifications defined by the RM-ODP.

This International Standard defines a method for automating the counting of Function Points that is generally consistent with the Function Point Counting Practices Manual, Release 4.3.1 (IFPUG CPM) produced by the International Function Point Users Group (IFPUG). Guidelines in this International Standard may differ from those in the IFPUG CPM at points where subjective judgments have to be replaced by the rules needed for automation. The IFPUG CPM was selected as the anchor for this International Standard because it is the most widely used functional measurement specification with a large supporting infrastructure maintained by a professional organization.

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

Information technology — Object management group — Interface definition language (IDL) 4.2

1 Scope

1.1 Overview

This International Standard specifies the OMG Interface Definition Language (IDL). IDL is a descriptive language used to define data types and interfaces in a way that is independent of the programming language or operating system/processor platform.

The IDL specifies only the syntax used to define the data types and interfaces. It is normally used in connection with other standards that further define how these types/interfaces are utilized in specific contexts and platforms:

- Separate "language mapping" standards define how the IDL-defined constructs map to specific programming languages, such as, C/C++, Java, C#, etc.
- Separate "serialization" standards define how data objects and method invocations are serialized into a format suitable for network transmission.
- Separate "middleware" standards, such as, DDS or CORBA leverage the IDL to define data-types, services, and interfaces.

The description of IDL grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF).

2 Conformance Criteria

This International Standard defines IDL such that it can be referenced by other standards. It contains no independent conformance points. It is up to the standards that depend on this International Standard to define their own conformance criteria. However, the general organization of the clauses (by means of atomic building blocks and profiles that group them) is intended to ease conformance description and scoping. That means that no standard using IDL 4.0 will be forced to be compliant with IDL constructs that are not relevant in its usage of IDL.

Conformance to this International Standard must follow these rules:

1. Future standards that use IDL shall reference this IDL International Standard or a future revision thereof.
2. Future revisions of current standards that use IDL may reference this IDL International Standard or a future revision thereof.
3. Reference to this International Standard shall result in a selection of building blocks possibly complemented by groups of annotations.

   a. All selected building blocks shall be supported entirely.

   b. Selected annotations shall be either supported as described in 8.2.2 Rules for Using Standardized Annotations, or fully ignored. In the latter case, the IDL-dependent standard shall not define a specific annotation, either with the same name and another meaning or with the same meaning and another name.

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

3 Normative References

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

- ISO/IEC 14882:2003, *Information Technology — Programming languages — C++*
- [RFC2119] IETF RFC 2119, "*Key words for use in RFCs to Indicate Requirement Levels*", S. Bradner, March 1997. Available from http://ietf.org/rfc/rfc2119
- [CORBA] Common Object Request Broker Architecture. OMG specification: formal/2012-11-12 (part1), formal/2012-11-14 (part2), formal/2012-11-16 (part3)

The following referenced documents were used as input to this International Standard:

- [DDS-XTypes]. Extensible and Dynamic Topic Types for DDS, Version 1.2. Available from: http://www.omg.org/spec/DDS-XTypes/1.2
- [DDS]. Data Distribution Service, Version 1.4. Available from: http://www.omg.org/spec/DDS/1.4
- [DDS-RPC]. Remote Procedure Call over DDS, Version 1.0. Available from: http://www.omg.org/spec/DDS-RPC/1.0

4 Terms and Definitions

In this International Standard:

- A building block is a consistent set of IDL rules that together form a piece of IDL functionality. Building blocks are atomic, meaning that if selected, they must be totally supported. Building blocks are described in Clause 7, IDL Syntax and Semantics.
- A group of annotations is a consistent set of annotations, expressed in IDL. Groups of annotations are described in Clause 8, Standardized Annotations.
- A profile is a selection of building blocks possibly complemented with groups of annotations that determines a specific IDL usage. Profiles are described in Clause 9, Profiles.

5 Symbols

The following abbreviations are used throughout this International Standard.

Acronym	Meaning
ASCII	American Standard Code for Information Interchange
BIPM	Bureau International des Poids et Mesures
CCM	CORBA Component Model
CORBA	Common Object Request Broker Architecture
DDS	Data Distribution Service
EBNF	Extended Backus Naur Form

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

Acronym	Meaning
IDL	Interface Definition Language
ISO	International Organization for Standardization
LwCCM	Lightweight CCM
OMG	Object Management Group
ORB	Object Request Broker
XTypes	eXtensible and dynamic topic Types (for DDS)

6 Additional Information

6.1 Acknowledgments

The following companies submitted this International Standard:

— Thales

— RTI

The following companies supported this International Standard:

— Mitre

— Northrop Grumman

— Remedy IT

6.2 History

Historically, IDL was designed to specify CORBA interfaces and subsequently CORBA components. For this reason the IDL standard was embedded in the CORBA documentation. However its expressive power made it very suitable for defining non-CORBA interfaces and data types. Consequently, it was used in the DDS standard and extended to support that usage. In recognition of these new usages, and expected future ones, IDL was separated into its own stand-alone standard, independent of its use by specific middleware technologies.

This International Standard completes the definition of IDL as a separate standard, an effort started with IDL 3.5.

IDL 3.5 gathered in a single standard all the CORBA-dedicated IDL, formerly specified as a collection of clauses within the CORBA 3 standard.

IDL 4.0 extended that corpus with the other source for IDL definitions, namely "Extensible and Dynamic Topic Types for DDS" in order to group all IDL constructs in a single comprehensive standard. It organized the IDL description into modular "Building Blocks" so that different levels of compliance are easier to specify and that future evolutions, if needed, can be made without side-effects on existing IDL usages.

IDL 4.1 improved the definition of the bitset type, added a bitmap type, and resolved some inconsistencies in the grammar.

IDL 4.2 added support for 8-bit integer types, added size-explicit keywords for integer types, enhanced the readability, and reordered the building blocks to follow a logical dependency progression.

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

7 IDL Syntax and Semantics

7.1 Overview

This clause describes OMG Interface Definition Language (IDL) middleware[1]-agnostic semantics[2] and defines the syntax for IDL grammatical constructs.

OMG IDL is a language that allows unambiguous specification of the interfaces that client objects[3] may use and (server) object implementations provide as well as all needed related constructs such as exceptions and data types. Data types are needed to specify parameters and return value of interfaces' operations. They can be used also as first class constructs.

IDL is a purely descriptive language. This means that actual programs that use these interfaces or create the associated data types cannot be written in IDL, but in a programming language, for which mappings from IDL constructs have been defined. The mapping of IDL constructs to a programming language will depend on the facilities available in that programming language. For example, an IDL exception might be mapped to a structure in a language that has no notion of exceptions, or to an exception in a language that does. The binding of IDL constructs to several programming languages is described in separate standards.

The clause is organized as follows:

- The description of IDL's lexical conventions is presented in 7.2 Lexical Conventions.
- A description of IDL preprocessing is presented in 7.3 Preprocessing
- The grammar itself is presented in 7.4 IDL Grammar
- The scoping rules for identifiers in an IDL standard are described in 7.5.

IDL-specific pragmas may appear anywhere in a standard; the textual location of these pragmas may be semantically constrained by a particular implementation.

A source file containing specifications written in IDL shall have a "**.idl**" extension.

The description of IDL grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF). However, to allow composition of specific parts of the description, while avoiding redundancy; a new operator (**::+**) has been added. This operator allows adding alternatives to an existing definition. For example, assuming the rule **x ::= y**, the rule **x ::+ z** shall be interpreted as **x ::= y | z**.

Table 7.1 lists the symbols used in this EBNF format and their meaning.

[1] In this document the word *middleware* refers to any piece of software that will make use of IDL-derived artifacts. CORBA and DDS implementations are examples of middleware. The word *compiler* refers to any piece of software that produces these IDL-derived artifacts based on an IDL specification.

[2] I.e., abstract semantics that is applicable to all IDL usages. When needed, middleware-specific interpretations of that abstract semantics will be given afterwards in dedicated clauses.

[3] Accordingly, *client objects* should be understood here as abstract clients, i.e., entities invoking operations provided by object implementations, regardless of the means used to perform this invocation or even whether those implementations are co-located or remotely accessible.

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

Table 7.1 — IDL EBNF

Symbol	Meaning
`::=`	Is defined to be *(left part of the rule is defined to be right part of the rule)*
`|`	Alternatively
`::+`	Is added as alternative *(left part of the rule is completed with right part of the rule as a new alternative)*
`<text>`	Nonterminal
`"text"`	Literal
`*`	The preceding syntactic unit can be repeated zero or more times
`+`	The preceding syntactic unit must be repeated at least once
`{}`	The enclosed syntactic units are grouped as a single syntactic unit
`[]`	The enclosed syntactic unit is optional – may occur zero or one time

7.2 Lexical Conventions

This sub clause[4] presents the lexical conventions of IDL. It defines tokens in an IDL specification and describes comments, identifiers, keywords, and literals - integer, character, and floating point constants and string literals.

An IDL specification logically consists of one or more files. A file is conceptually translated in several phases.

The first phase is preprocessing, which performs file inclusion and macro substitution. Preprocessing is controlled by directives introduced by lines having # as the first character other than white space. The result of preprocessing is a sequence of tokens. Such a sequence of tokens, that is, a file after preprocessing, is called a translation unit.

IDL uses the ASCII character set, except for string literals and character literals, which use the ISO Latin-1 (8859-1) character set. The ISO Latin-1 character set is divided into alphabetic characters (letters) digits, graphic characters, the space (blank) character, and formatting characters. Table 7.2 shows the ISO Latin-1 alphabetic characters; upper and lower case equivalences are paired. The ASCII alphabetic characters are shown in the left-hand column of Table 7.2.

Table 7.2 — Characters

Char.	Description	Char.	Description
Aa	Upper/Lower-case A	**Àà**	Upper/Lower-case A with grave accent
Bb	Upper/Lower-case B	**Áá**	Upper/Lower-case A with acute accent
Cc	Upper/Lower-case C	**Ââ**	Upper/Lower-case A with circumflex accent
Dd	Upper/Lower-case D	**Ãã**	Upper/Lower-case A with tilde
Ee	Upper/Lower-case E	**Ää**	Upper/Lower-case A with dieresis
Ff	Upper/Lower-case F	**Åå**	Upper/Lower-case A with ring above
Gg	Upper/Lower-case G	**Ææ**	Upper/Lower-case dipthong A with E
Hh	Upper/Lower-case H	**Çç**	Upper/Lower-case C with cedilla
Ii	Upper/Lower-case I	**Èè**	Upper/Lower-case E with grave accent

[4] This sub clause is an adaptation of The Annotated C++ Reference Manual, Clause 2; it differs in the list of legal keywords and punctuation.

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

Char.	Description	Char.	Description
Jj	Upper/Lower-case J	Éé	Upper/Lower-case E with acute accent
Kk	Upper/Lower-case K	Êê	Upper/Lower-case E with circumflex accent
Ll	Upper/Lower-case L	Ëë	Upper/Lower-case E with dieresis
Mm	Upper/Lower-case M	Ìì	Upper/Lower-case I with grave accent
Nn	Upper/Lower-case N	Íí	Upper/Lower-case I with acute accent
Oo	Upper/Lower-case O	Îî	Upper/Lower-case I with circumflex accent
Pp	Upper/Lower-case P	Ïï	Upper/Lower-case I with dieresis
Qq	Upper/Lower-case Q	Ññ	Upper/Lower-case N with tilde
Rr	Upper/Lower-case R	Òò	Upper/Lower-case O with grave accent
Ss	Upper/Lower-case S	Óó	Upper/Lower-case O with acute accent
Tt	Upper/Lower-case T	Ôô	Upper/Lower-case O with circumflex accent
Uu	Upper/Lower-case U	Õõ	Upper/Lower-case O with tilde
Vv	Upper/Lower-case V	Öö	Upper/Lower-case O with dieresis
Ww	Upper/Lower-case W	Øø	Upper/Lower-case O with oblique stroke
Xx	Upper/Lower-case X	Ùù	Upper/Lower-case U with grave accent
Yy	Upper/Lower-case Y	Úú	Upper/Lower-case U with acute accent
Zz	Upper/Lower-case Z	Ûû	Upper/Lower-case U with circumflex accent
		Üü	Upper/Lower-case U with dieresis
		ß	Lower-case German sharp S
		ÿ	Lower-case Y with dieresis

Table 7.3 lists the decimal digit characters.

Table 7.3 — Decimal Digits

0	1	2	3	4	5	6	7	8	9

Table 7.4 shows the graphic characters.

Table 7.4 — Graphic Characters

Char.	Description	Char.	Description
!	exclamation point	¡	inverted exclamation mark
"	double quote	¢	cent sign
#	number sign	£	pound sign
$	dollar sign	¤	currency sign
%	percent sign	¥	yen sign
&	ampersand	¦	broken bar

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

Char.	Description	Char.	Description
'	apostrophe	§	section/paragraph sign
(	left parenthesis	¨	dieresis
)	right parenthesis	©	copyright sign
*	asterisk	ª	feminine ordinal indicator
+	plus sign	«	left angle quotation mark
,	comma	¬	not sign
-	hyphen, minus sign		soft hyphen
.	period, full stop	®	registered trade mark sign
/	solidus	¯	macron
:	colon	°	ring above, degree sign
;	semicolon	±	plus-minus sign
<	less-than sign	²	superscript two
=	equals sign	³	superscript three
>	greater-than sign	´	acute
?	question mark	µ	micro
@	commercial at	¶	pilcrow
[	left square bracket	·	middle dot
\	reverse solidus	¸	cedilla
]	right square bracket	¹	superscript one
^	circumflex	º	masculine ordinal indicator
_	low line, underscore	»	right angle quotation mark
`	grave	¼	vulgar fraction 1/4
{	left curly bracket	½	vulgar fraction 1/2
|	vertical line	¾	vulgar fraction 3/4
}	right curly bracket	¿	inverted question mark
~	tilde	×	multiplication sign
		÷	division sign

The formatting characters are shown in Table 7.5.

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

Table 7.5 — Formatting Characters

Description	Abbreviation	ISO 646 Octal Value
alert	BEL	007
backspace	BS	010
horizontal tab	HT	011
newline	NL, LF	012
vertical tab	VT	013
form feed	FF	014
carriage return	CR	015

7.2.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators.

Blanks, horizontal and vertical tabs, newlines, form feeds, and comments (collective, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

7.2.2 Comments

The characters /* start a comment, which terminates with the characters */. These comments do not nest.

The characters // start a comment, which terminates at the end of the line on which they occur.

The comment characters //, /*, and */ have no special meaning within a // comment and are treated just like other characters. Similarly, the comment characters // and /* have no special meaning within a /* comment.

Comments may contain alphabetic, digit, graphic, space, horizontal tab, vertical tab, form feed, and newline characters.

7.2.3 Identifiers

An identifier is an arbitrarily long sequence of ASCII alphabetic, digit, and underscore (_) characters. The first character must be an ASCII alphabetic character. All characters are significant.

IDL identifiers are case insensitive. However, all references to a definition must use the same case as the defining occurrence. This allows natural mappings to case-sensitive languages.

7.2.3.1 Collision Rules

When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. Table 7.2 defines the equivalence mapping of upper- and lower-case letters.
- All characters are significant.

Identifiers that differ only in case collide, and will yield a compilation error under certain circumstances. An identifier for a given definition must be spelled identically (e.g., with respect to case) throughout a specification.

There is only one namespace for IDL identifiers in each scope. Using the same identifier for a constant and an interface, for example, produces a compilation error.

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

EXAMPLE

```
module M {
    typedef long Foo;
    const long thing = 1;
    interface thing {                        // Error: reuse of identifier thing
        void doit (
                in Foo foo                   // Error: Foo and foo collide…
                );                           //        … and refer to different things
        readonly attribute long Attribute;   // Error: Attribute collides with keyword…
                                             //        … attribute
        };
    };
```

7.2.3.2 Escaped Identifiers

As all languages, IDL uses some reserved words called *keywords* (see 7.2.4).

As IDL evolves, new keywords that are added to the IDL language may inadvertently collide with identifiers used in existing IDL and programs that use that IDL. Fixing these collisions will require not only the IDL to be modified, but programming language code that depends upon that IDL will have to change as well. The language mapping rules for the renamed IDL identifiers will cause the mapped identifier names (e.g., method names) to be changed.

To minimize the amount of work, users may lexically "escape" identifiers by prepending an underscore (_) to an identifier. *This is a purely lexical convention that ONLY turns off keyword checking*. The resulting identifier follows all the other rules for identifier processing. For example, the identifier **_AnIdentifier** is treated as if it were **AnIdentifier**.

EXAMPLE

```
module M {
        interface thing {
                attribute boolean abstract;          // Error: abstract collides with keyword abstract
                attribute boolean _abstract;         // OK: abstract is an identifier
                };
        };
```

NOTE To avoid unnecessary confusion for readers of IDL, it is recommended that IDL specifications only use the escaped form of identifiers when the non-escaped form clashes with a newly introduced IDL keyword. It is also recommended that interface designers avoid defining new identifiers that are known to require escaping. Escaped literals are only recommended for IDL that expresses legacy items, or for IDL that is mechanically generated.

7.2.4 Keywords

The identifiers listed in Table 7.6 are reserved for use as keywords and may not be used for another purpose, unless escaped with a leading underscore.

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

Table 7.6 — All IDL Keywords

abstract	any	alias	attribute	bitfield
bitmask	bitset	boolean	case	char
component	connector	const	consumes	context
custom	default	double	exception	emits
enum	eventtype	factory	FALSE	finder
fixed	float	getraises	home	import
in	inout	interface	local	long
manages	map	mirrorport	module	multiple
native	Object	octet	oneway	out
primarykey	private	port	porttype	provides
public	publishes	raises	readonly	setraises
sequence	short	string	struct	supports
switch	TRUE	truncatable	typedef	typeid
typename	typeprefix	unsigned	union	uses
ValueBase	valuetype	void	wchar	wstring
int8	uint8	int16	int32	int64
uint16	uint32	uint64		

Keywords must be written exactly as shown in the above list. Identifiers that collide with keywords (see 7.2.3) are illegal. For example, **boolean** is a valid keyword; **Boolean** and **BOOLEAN** are illegal identifiers.

EXAMPLE

```
module M {
        typedef Long Foo;                   // Error: keyword is long not Long
        typedef boolean BOOLEAN;            // Error: BOOLEAN collides with the keyword…
                                            //        …boolean;
};
```

NOTE As the IDL grammar is now organized in building blocks that dedicated profiles may include or not, some of these keywords may be irrelevant to some profiles. Each building block description (see 7.4) includes the set of keywords that are specific to it. The minimum set of keywords for a given profile results from the union of all the ones of the included building blocks. However, to avoid unnecessary confusion for readers of IDL and to allow IDL compilers supporting several profiles in a single tool, it is recommended that IDL specifications avoid using any of the keywords listed in Table 7.6.

7.2.5 Other Characters Recognized by IDL

IDL specifications use the characters shown in Table 7.7 as punctuation.

Table 7.7 — Punctuation

;	{	}	:	,	=	+	−	(	)	<	>	[	]
'	"	\	|	^	&	*	/	%	~	@			

In addition, the tokens listed in Table 7.8 are used by the preprocessor.

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

Table 7.8 — Tokens

#	##	!	||	&&

7.2.6 Literals

This sub clause describes the following literals:

- Integer
- Character
- Floating-point
- String
- Fixed-point

7.2.6.1 Integer Literals

An integer literal consisting of a sequence of digits is taken to be *decimal* (base ten) unless it begins with **0** (digit zero).

A sequence of digits starting with **0** is taken to be an *octal integer* (base eight). The digits **8** and **9** are not octal digits and thus are not allowed in an octal integer literal.

A sequence of digits preceded by **0x** (or **0X**) is taken to be a *hexadecimal integer* (base sixteen). The hexadecimal digits include **a** (or **A**) through **f** (or **F**) with decimal values ten through fifteen, respectively.

For example, the number twelve can be written **12**, **014**, or **0XC**.

7.2.6.2 Character Literals

7.2.6.2.1 Wide and Non-wide Characters

Character literals may have type **char** (non-wide character) or **wchar** (wide character).

Both wide and non-wide character literals must be specified using characters from the ISO Latin-1 (8859-1) character set.

A **char** is an 8-bit quantity with a numerical value between 0 and 255 (decimal). The value of a space, alphabetic, digit, or graphic character literal is the numerical value of the character as defined in the ISO Latin-1 (8859-1) character set standard (see Table 7.2, Table 7.3, and Table 7.4). The value of a **null** is 0. The value of a formatting character literal is the numerical value of the character as defined in the ISO 646 standard (see Table 7.5). The meaning of all other characters is implementation-dependent.

A **wchar** (wide character) is intended to encode wide characters from any character set. Its size is implementation dependent.

A character literal is one or more characters enclosed in single quotes, as in:

```
const char C1 = 'X';
```

Wide character literals have in addition an **L** prefix, as in:

```
const wchar C2 = L'X';
```

Attempts to assign a wide character literal to a non-wide character constant, or to assign a non-wide character literal to a wide character constant, shall be treated as an error.

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

7.2.6.2.2 Escape Sequences to Represent Character Literals

Nongraphic characters shall be represented using escape sequences as defined below in Table 7.9. Note that escape sequences shall be used to represent single quote and backslash characters in character literals.

Table 7.9 — Escape Sequences

Description	Escape Sequence
newline	`\n`
horizontal tab	`\t`
vertical tab	`\v`
backspace	`\b`
carriage return	`\r`
form feed	`\f`
alert	`\a`
backslash	`\\`
question mark	`\?`
single quote	`\'`
double quote	`\"`
octal number	`\ooo`
hexadecimal number	`\xhh`
Unicode character	`\uhhhh`

If the character following a backslash is not one of those specified, the behavior is undefined. An escape sequence specifies a single character.

The escape **\ooo** consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape **\xhh** consists of the backslash followed by **x** followed by one or two hexadecimal digits that are taken to specify the value of the desired character.

A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character constant is implementation dependent if it exceeds that of the largest char.

The escape **\uhhhh** consists of a backslash followed by the character **u**, followed by one, two, three, or four hexadecimal digits. This represents a Unicode character literal. For example, the literal **\u002E** represents the Unicode period '.' character and the literal **\u3BC** represents the Unicode Greek small letter 'μ' (mu). The **\u** escape is valid only with **wchar** and **wstring** types. Because a wide string literal is defined as a sequence of wide character literals, a sequence of **\u** literals can be used to define a wide string literal.

Attempts to set a char type to a **\u** defined literal or a string type to a sequence of **\u** literals result in an error.

7.2.6.3 String Literals

Strings are null-terminated sequences of characters. Strings are of type **string** if they are made of non-wide characters or **wstring** (wide string) if they are made of wide characters.

A string literal is a sequence of character literals with the exception of the character with numeric value 0, surrounded by double quotes, as in:

```
const string S1 = "Hello";
```


© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

Wide string literals have in addition an **L** prefix, for example:

```
const wstring S2 = L"Hello";
```

Both wide and non-wide string literals must be specified using characters from the ISO Latin-1 (8859-1) character set. A string literal shall not contain the character '\0'. A wide string literal shall not contain the wide character with value zero.

Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For example, **"\xA" "B"** contains the two characters '**xA**' and '**B**' after concatenation (and not the single hexadecimal character **'\xAB'**).

The size of a string literal is the number of character literals enclosed by the quotes, after concatenation. Within a string, the double quote character **"** must be preceded by a \.

Attempts to assign a wide string literal to a non-wide string constant or to assign a non-wide string literal to a wide string constant result in a compile-time diagnostic.

7.2.6.4 Floating-point Literals

A floating-point literal consists of an integer part, a decimal point (.), a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (but not both) may be missing; either the decimal point or the letter **e** (or **E**) and the exponent (but not both) may be missing.

7.2.6.5 Fixed-Point Literals

A fixed-point decimal literal consists of an integer part, a decimal point (.), a fraction part and a **d** or **D**. The integer and fraction parts both consist of a sequence of decimal (base 10) digits. Either the integer part or the fraction part (but not both) may be missing; the decimal point (but not the letter **d** or **D**) may be missing.

7.3 Preprocessing

IDL shall be preprocessed according to the specification of the preprocessor in ISO/IEC 14882:2003. The preprocessor may be implemented as a separate process or built into the IDL compiler.

Lines beginning with **#** (also called "directives") communicate with this preprocessor. White space may appear before the **#**. These lines have syntax independent of the rest of IDL; they may appear anywhere and have effects that last (independent of the IDL scoping rules) until the end of the translation unit. The textual location of IDL-specific pragmas may be semantically constrained.

A preprocessing directive (or any line) may be continued on the next line in a source file by placing a backslash character (\), immediately before the newline at the end of the line to be continued. The preprocessor effects the continuation by deleting the backslash and the newline before the input sequence is divided into tokens. A backslash character may not be the last character in a source file.

A preprocessing token is an IDL token (see 7.2.1), a file name as in a **#include** directive, or any single character other than white space that does not match another preprocessing token.

The primary use of the preprocessing facilities is to include definitions from other IDL specifications. Text in files included with a **#include** directive is treated as if it appeared in the including file.

NOTE Generating code for included files is an IDL compiler implementation-specific issue. To support separate compilation, IDL compilers may not generate code for included files, or do so only if explicitly instructed.

7.4 IDL Grammar

The grammar for a well-formed IDL specification is described by rules expressed in Extended Backus Naur Form (EBNF) completed to support rule extensions as explained in 7.1. Table 7.1 gathers all the symbols used in rules.

These rules are grouped in atomic *building blocks* that will be themselves grouped to form dedicated *profiles*. Atomic means that they cannot be split (in other words a given profile will contain or not a given building block, but never just parts of it).

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

In all the building block descriptions, the normative rules are grouped in a sub clause entitled "Syntax" and written in **Arial bold**. They are then detailed in a sub clause entitled "Explanations and Semantics", where they are copied to ease understanding. These copies are actually hyperlinks to the originals and are written in ***Arial bold-italics***.

In all the rules, the following non-terminals pre-exist and are not detailed.

Table 7.10 — List of pre-existing non-terminals in IDL rules

Token	Explanation
<identifier>	A valid identifier, as explained in 7.2.3 Identifiers
<integer_literal>	A valid integer literal as explained in7.2.6.1, Integer Literals
<string_literal>	A valid string literal as explained in 7.2.6.3 String Literals
<wide_string_literal>	A valid wide string literal as explained in 7.2.6.3, String Literals
<character_literal>	A valid character literal as explained in 7.2.6.2, Character Literals
<wide_character_literal>	A valid wide character literal as explained in7.2.6.2, Character Literals
<fixed_pt_literal>	A valid fixed point literal as explained in 7.2.6.5 Fixed-Point Literals
<floating_pt_literal>	A valid floating point literal as explained in 7.2.6.4 Floating Point Literals

7.4.1 Building Block Core Data Types

7.4.1.1 Purpose

This building block constitutes the core of any IDL specialization (all other building blocks assume that this one is included). It contains the syntax rules that allow defining most data types and the syntax rules that allow IDL basic structuring (i.e., modules). Since it is the only mandatory building block, it also contains the root nonterminal <**specification**> for the grammar itself.

7.4.1.2 Dependencies with other Building Blocks

This building block is the root for all other building blocks and requires no other ones.

7.4.1.3 Syntax

The following set of rules form the building block:

```
(1)  <specification>     ::=  <definition>
(2)  <definition>        ::=  <module_dcl> ";"
                         |    <const_dcl> ";"
                         |    <type_dcl> ";"
(3)  <module_dcl>        ::=  "module" <identifier> "{" <definition>+ "}"
(4)  <scoped_name>       ::=  <identifier>
                         |    "::" <identifier>
                         |    <scoped_name> "::" <identifier>
(5)  <const_dcl>         ::=  "const" <const_type> <identifier> "=" <const_expr>
(6)  <const_type>        ::=  <integer_type>
                         |    <floating_pt_type>
                         |    <fixed_pt_const_type>
                         |    <char_type>
```


© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

```
                                  |       <wide_char_type>
                                  |       <boolean_type>
                                  |       <octet_type>
                                  |       <string_type>
                                  |       <wide_string_type>
                                  |       <scoped_name>
(7)  <const_expr>                 ::=     <or_expr>
(8)  <or_expr>                    ::=     <xor_expr>
                                  |       <or_expr> "|" <xor_expr>
(9)  <xor_expr>                   ::=     <and_expr>
                                  |       <xor_expr> "^" <and_expr>
(10)  <and_expr>                  ::=     <shift_expr>
                                  |       <and_expr> "&" <shift_expr>
(11) <shift_expr>                 ::=     <add_expr>
                                  |       <shift_expr> ">>" <add_expr>
                                  |       <shift_expr> "<<" <add_expr>
(12) <add_expr>                   ::=     <mult_expr>
                                  |       <add_expr> "+" <mult_expr>
                                  |       <add_expr> "-" <mult_expr>
(13) <mult_expr>                  ::=     <unary_expr>
                                  |       <mult_expr> "*" <unary_expr>
                                  |       <mult_expr> "/" <unary_expr>
                                  |       <mult_expr> "%" <unary_expr>
(14) <unary_expr>                 ::=     <unary_operator> <primary_expr>
                                  |       <primary_expr>
(15) <unary_operator>             ::=     "-"
                                  |       "+"
                                  |       "~"
(16) <primary_expr>               ::=     <scoped_name>
                                  |       <literal>
                                  |       "(" <const_expr> ")"
(17) <literal>                    ::=     <integer_literal>
                                  |       <floating_pt_literal>
                                  |       <fixed_pt_literal>
                                  |       <character_literal>
                                  |       <wide_character_literal>
                                  |       <boolean_literal>
                                  |       <string_literal>
                                  |       <wide_string_literal>
(18) <boolean_literal>            ::=     "TRUE"
                                  |       "FALSE"
(19) <positive_int_const>         ::=     <const_expr>
(20) <type_dcl>                   ::=     <constr_type_dcl>
                                  |       <native_dcl>
                                  |       <typedef_dcl>
(21) <type_spec>                  ::=     <simple_type_spec>
(22) <simple_type_spec>           ::=     <base_type_spec>
                                  |       <scoped_name>
(23) <base_type_spec>             ::=     <integer_type>
                                  |       <floating_pt_type>
                                  |       <char_type>
                                  |       <wide_char_type>
                                  |       <boolean_type>
                                  |       <octet_type>
(24) <floating_pt_type>           ::=     "float"
                                  |       "double"
                                  |       "long" "double"
(25) <integer_type>               ::=     <signed_int>
                                  |       <unsigned_int>
(26) <signed_int>                 ::=     <signed_short_int>
                                  |       <signed_long_int>
                                  |       <signed_longlong_int>
(27) <signed_short_int>           ::=     "short"
(28) <signed_long_int>            ::=     "long"
(29) <signed_longlong_int>        ::=     "long" "long"
(30) <unsigned_int>               ::=     <unsigned_short_int>
                                  |       <unsigned_long_int>
```

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

```
                                  |      <unsigned_longlong_int>
(31) <unsigned_short_int>         ::=    "unsigned" "short"
(32) <unsigned_long_int>          ::=    "unsigned" "long"
(33) <unsigned_longlong_int>      ::=    "unsigned" "long" "long"
(34) <char_type>                  ::=    "char"
(35) <wide_char_type>             ::=    "wchar"
(36) <boolean_type>               ::=    "boolean"
(37) <octet_type>                 ::=    "octet"
(38) <template_type_spec>         ::=    <sequence_type>
                                  |      <string_type>
                                  |      <wide_string_type>
                                  |      <fixed_pt_type>
(39) <sequence_type>              ::=    "sequence" "<" <type_spec> "," <positive_int_const> ">"
                                  |      "sequence" "<" <type_spec> ">"
(40) <string_type>                ::=    "string" "<" <positive_int_const> ">"
                                  |      "string"
(41) <wide_string_type>           ::=    "wstring" "<" <positive_int_const> ">"
                                  |       "wstring"
(42) <fixed_pt_type>              ::=    "fixed" "<" <positive_int_const> "," <positive_int_const> ">"
(43) <fixed_pt_const_type>        ::=    "fixed"
(44) <constr_type_dcl>            ::=    <struct_dcl>
                                  |      <union_dcl>
                                  |      <enum_dcl>
(45) <struct_dcl>                 ::=    <struct_def>
                                  |      <struct_forward_dcl>
(46) <struct_def>                 ::=    "struct" <identifier> "{" <member>+ "}"
(47) <member>                     ::=    <type_spec> <declarators> ";"
(48) <struct_forward_dcl>         ::=    "struct" <identifier>
(49) <union_dcl>                  ::=    <union_def>
                                  |      <union_forward_dcl>
(50) <union_def>                  ::=    "union" <identifier> "switch" "(" <switch_type_spec> ")"
                                         "{" <switch_body> "}"
(51) <switch_type_spec>           ::=    <integer_type>
                                  |      <char_type>
                                  |      <boolean_type>
                                  |      <scoped_name>
(52) <switch_body>                ::=    <case>+
(53) <case>                       ::=    <case_label>+ <element_spec> ";"
(54) <case_label>                 ::=    "case" <const_expr> ":"
                                  |      "default" ":"
(55) <element_spec>               ::=    <type_spec> <declarator>
(56) <union_forward_dcl>          ::=    "union" <identifier>
(57) <enum_dcl>                   ::=    "enum" <identifier>
                                         "{" <enumerator> { "," <enumerator> } * "}"
(58) <enumerator>                 ::=    <identifier>
(59) <array_declarator>           ::=    <identifier> <fixed_array_size>+
(60) <fixed_array_size>           ::=    "[" <positive_int_const> "]"
(61) <native_dcl>                 ::=    "native" <simple_declarator>
(62) <simple_declarator>          ::=    <identifier>
(63) <typedef_dcl>                ::=    "typedef" <type_declarator>
(64) <type_declarator>            ::=    { <simple_type_spec>
                                         | <template_type_spec>
                                         | <constr_type_dcl>
                                         } <any_declarators>
(65) <any_declarators>            ::=    <any_declarator> { "," <any_declarator> }*
(66) <any_declarator>             ::=    <simple_declarator>
                                  |      <array_declarator>
(67) <declarators>                ::=    <declarator> { "," <declarator> }*
(68) <declarator>                 ::=    <simple_declarator>
```

7.4.1.4 Explanations and Semantics

7.4.1.4.1 IDL Specification

An IDL specification consists of one or more definitions.

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

```
(1) <specification>     ::=     <definition>+
```

In this building block, supported definitions are: module definitions, constant definitions and (data) type definitions as expressed in the following rule:

```
(2)     <definition>            ::= <module_dcl> ";"
                                |   <const_dcl> ";"
                                |   <type_dcl> ";"
```

7.4.1.4.2 Modules

A module is a grouping construct. Its definition satisfies the following rule:

```
(3)     <module_dcl>            ::= "module" <identifier> "{" <definition>+ "}"
```

A module is declared with:

- The **module** keyword.
- An identifier (<**identifier**>) which is the name of the module. That name is then used to scope embedded IDL identifiers.
- A list of at least one definition (<**definition**>+) enclosed within braces ({}). Those definitions form the module body.

A scoped name is built according to the following rule:

```
(4)     <scoped_name>           ::= <identifier>
                                |   "::" <identifier>
                                |   <scoped_name> "::" <identifier>
```

For more details on scoping rules, see 7.5.

An IDL module can be reopened, which means that when a module declaration is encountered with a name already given to an existing module, all the enclosed definitions are appended to that existing module: the two module statements are thus considered as subsequent parts of the same module description.

7.4.1.4.3 Constants

Well-formed constants shall follow the following rules:

```
(5)     <const_dcl>             ::= "const" <const_type> <identifier> "=" <const_expr>

(6)     <const_type>            ::= <integer_type>
                                |   <floating_pt_type>
                                |   <fixed_pt_const_type>
                                |   <char_type>
                                |   <wide_char_type>
                                |   <boolean_type>
                                |   <octet_type>
                                |   <string_type>
                                |   <wide_string_type>
                                |   <scoped_name>

(7)     <const_expr>            ::= <or_expr>

(8)     <or_expr>               ::= <xor_expr>
                                |   <or_expr> "|" <xor_expr>

(9)     <xor_expr>              ::= <and_expr>
                                |   <xor_expr> "^" <and_expr>

(10)    <and_expr>              ::= <shift_expr>
                                |   <and_expr> "&" <shift_expr>

(11)    <shift_expr>            ::= <add_expr>
                                |   <shift_expr> ">>" <add_expr>
                                |   <shift_expr> "<<" <add_expr>
```

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

```
(12)  <add_expr>          ::= <mult_expr>
                          |   <add_expr> "+" <mult_expr>
                          |   <add_expr> "-" <mult_expr>

(13)  <mult_expr>         ::= <unary_expr>
                          |   <mult_expr> "*" <unary_expr>
                          |   <mult_expr> "/" <unary_expr>
                          |   <mult_expr> "%" <unary_expr>

(14)  <unary_expr>        ::= <unary_operator> <primary_expr>
                          |   <primary_expr>

(15)  <unary_operator>    ::= "-"
                          |   "+"
                          |   "~"

(16)  <primary_expr>      ::= <scoped_name>
                          |   <literal>
                          |   "(" <const_expr> ")"

(17)  <literal>           ::= <integer_literal>
                          |   <floating_pt_literal>
                          |   <fixed_pt_literal>
                          |   <character_literal>
                          |   <wide_character_literal>
                          |   <boolean_literal>
                          |   <string_literal>
                          |   <wide_string_literal>

(18)  <boolean_literal>   ::= "TRUE"
                          |   "FALSE"

(19)  <positive_int_const> ::= <const_expr>
```

According to those rules, a constant is defined by:

- The **const** keyword.
- A type declaration, which shall denote a type suitable for a constant (**<const_type>**), i.e.,:
    - Either one of the following: **<integer_type>**, **<floating_pt_type>**, **<fixed_pt_const_type>**, **<char_type>**, **<wide_char_type>**, **<boolean_type>**, **<octet_type>**, **<string_type>**, **<wide_string_type>**, or a previously defined enumeration. For a definition of those types, see 7.4.1.4.4, Data Types.
    - Or a **<scoped_name>**, which shall be a previously defined name of one of the above.
- The name given to the constant (<**identifier**>).
- The operator =.
- A value expression (<**const_expr**>), which shall be consistent with the type declared for the constant.

For evaluating the value expression (right hand side of the constant declaration), the following rules shall be applied:

- If the type of an integer constant is **long** or **unsigned long**, then each sub-expression of the associated constant expression is treated as an **unsigned long** by default, or a signed long for negated literals or negative integer constants. It is an error if any sub-expression values exceed the precision of the assigned type (**long** or **unsigned long**), or if a final expression value (of type **unsigned long**) exceeds the precision of the target type (**long**).
- If the type of an integer constant is **long long** or **unsigned long long**, then each sub-expression of the associated constant expression is treated as an **unsigned long long** by default, or a signed **long long** for negated literals or negative integer constants. It is an error if any sub-expression values exceed the precision of the assigned type (**long long** or **unsigned long long**), or if a final expression value (of type **unsigned long long**) exceeds the precision of the target type (**long long**).
- If the type of a floating-point constant is double, then each sub-expression of the associated constant expression is treated as a **double**. It is an error if any sub-expression value exceeds the precision of **double**.

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

- If the type of a floating-point constant is **long double**, then each sub-expression of the associated constant expression is treated as a **long double**. It is an error if any sub-expression value exceeds the precision of **long double**.
- An infix operator may combine two integer types, floating point types or fixed point types, but not mixtures of these. Infix operators shall be applicable only to integer, floating point, and fixed point types.
- Integer expressions shall be evaluated based on the type of each argument of a binary operator in turn. If either argument is **unsigned long long**, it shall use **unsigned long long**. If either argument is **long long**, it shall use **long long**. If either argument is **unsigned long**, it shall use **unsigned long**. Otherwise it shall use **long**. The final result of an integer arithmetic expression shall fit in the range of the declared type of the constant; otherwise it shall be treated as an error. In addition to the integer types, the final result of an integer arithmetic expression may be assigned to an octet constant, subject to it fitting in the range for **octet** type.
- Floating point expressions shall be evaluated based on the type of each argument of a binary operator in turn. If either argument is **long double**, it shall use **long double**. Otherwise it shall use **double**. The final result of a floating point arithmetic expression shall fit in the range of the declared type of the constant; otherwise it shall be treated as an error.
- Fixed-point decimal constant expressions shall be evaluated as follows. A fixed-point literal has the apparent number of total and fractional digits. For example, **0123.450d** is considered to be **fixed<7,3>** and **3000.00d** is **fixed<6,2>.** Prefix operators do not affect the precision; a prefix + is optional, and does not change the result. The upper bounds on the number of digits and scale of the result of an infix expression, **fixed<d1 ,s1> op fixed<d2,s2>**, are shown in the following table.

Table 7.11 — Operations on fixed-point decimal constants

Op	Result: fixed<d,s>
+	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
-	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
*	fixed<d1+d2, s1+s2>
/	fixed<(d1-s1+s2) + sinf , sinf>

- A quotient may have an arbitrary number of decimal places, denoted by a scale of sinf. The computation proceeds pairwise, with the usual rules for left-to-right association, operator precedence, and parentheses. All intermediate computations shall be performed using double precision (i.e., 62 digits) arithmetic. If an individual computation between a pair of fixed-point literals actually generates more than 31 significant digits, then a 31-digit result is retained as follows:

  fixed<d,s> => fixed<31, 31-d+s>

- Leading and trailing zeros shall not be considered significant. The omitted digits shall be discarded; rounding shall not be performed. The result of the individual computation then proceeds as one literal operand of the next pair of fixed-point literals to be computed.
- Unary **(+ -)** and binary (*** / + -**) operators shall be applicable in floating-point and fixed-point expressions.
  - The **+** unary operator shall have no effect; the **–** unary operator indicates that the sign of the following expression is inverted.
  - The ***** binary operator indicates that the two operands shall be multiplied; the **/** binary operator indicates that the first operand shall be divided by the second one; the **+** binary operator indicates that the two operands shall be added; the **–** binary operator indicates that the second operand shall be subtracted from the first one.
- Unary (**+ - ~**) and binary (*** / % + - << >> & | ^**) operators are applicable in integer expressions.

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

- The **+** unary operator shall have no effect; the **–** unary operator indicates that the sign of the following expression is inverted.
- The ***** binary operator indicates that the two operands shall be multiplied; the **/** binary operator indicates that the first operand shall be divided by the second one; the **+** binary operator indicates that the two operands shall be added; the **–** binary operator indicates that the second operand shall be subtracted from the first one.
- The **~** unary operator indicates that the bit-complement of the expression to which it is applied shall be generated. For the purposes of such expressions, the values are 2's complement numbers. As such, the complement can be generated as follows:

Table 7.12 — 2's complement numbers

Integer Constant Expression Type	Generated 2's Complement Numbers
long	long -(value+1)
unsigned long	unsigned long (2**32-1) - value
long long	long long -(value+1)
unsigned long long	unsigned long (2**64-1) - value

- The **%** binary operator yields the remainder from the division of the first expression by the second. If the second operand of **%** is 0, the result is undefined; otherwise **(a/b)*b + a%b** is equal to **a**.
  If both operands are non-negative, then the remainder is non-negative; if not, the sign of the remainder is implementation dependent.
- The **<<** binary operator indicates that the value of the left operand shall be shifted left the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand shall be in the range 0 <= right operand < 64.
- The **>>** binary operator indicates that the value of the left operand shall be shifted right the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand shall be in the range 0 <= right operand < 64.
- The **&** binary operator indicates that the logical, bitwise **AND** of the left and right operands shall be generated.
- The **|** binary operator indicates that the logical, bitwise **OR** of the left and right operands shall be generated.
- The **^** binary operator indicates that the logical, bitwise **EXCLUSIVE-OR** of the left and right operands shall be generated.

- <**positive_int_const**> shall evaluate to a positive integer constant.

The consistency rules between the value (right hand side of the constant declaration) and the constant type declaration (left hand side) are as follows:

- Integer literals have positive integer values. Constant integer literals shall be considered **unsigned long** unless the value is too large, then they shall be considered **unsigned long long**. Unary minus shall be considered an operator, not a part of an integer literal. Only integer values can be assigned to integer type (**short, long, long long**) constants, and **octet** constants. Only positive integer values can be assigned to unsigned integer type constants. If the value of an integer constant declaration is too large to fit in the actual type of the constant on the left hand side (for example **const short s = 655592**;) or is inappropriate for the actual type of the constant (for example **const octet o = -54**;) it shall be treated as an error.
- Octet literals have integer value in the range 0…255. If the right hand side of an octet constant declaration is outside this range, it shall be treated as an error.

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

- An octet constant can be defined using an integer literal or an integer constant expression but values outside the range 0…255 shall be treated as an error.
- Floating point literals have floating point values. Only floating point values can be assigned to floating point type (**float, double, long double**) constants. Constant floating point literals are considered double unless the value is too large, then they are considered **long double**. If the value of the right hand side is too large to fit in the actual type of the constant to which it is being assigned, it shall be treated as an error. Truncation on the right for floating point types is OK.
- Fixed point literals have fixed point values. Only fixed point values can be assigned to fixed point type constants. If the fixed point value in the expression on the right hand side is too large to fit in the actual fixed point type of the constant on the left hand side, then it shall be treated as an error. Truncation on the right for fixed point types is OK.
- An **enum** constant can only be defined using a scoped name for the enumerator. The scoped name is resolved using the normal scope resolution rules (see 7.5). For example:

```
enum Color { red, green, blue };
const Color FAVORITE_COLOR = red;
module M {
        enum Size { small, medium, large };
        };
const M::Size MYSIZE = M::medium;
```

- The constant name for the value of an enumerated constant definition shall denote one of the enumerators defined for the enumerated type of the constant. For example:

```
const Color col = red;                  // is OK but
const Color another = M::medium;        // is an error
```

7.4.1.4.4 Data Types

A data type may be either a simple type or a constructed one. Those different kinds are detailed in the following clauses.

7.4.1.4.4.1 Referencing Types

Type declarations may reference other types. These type references can be split in several categories:

- References to basic types representing primitive builtin types such as numbers and characters. These use the keyword that identifies the type.
- References to types explicitly constructed or explictly named types. These use the scoped name of the type.
- References to anonymous template types that must be instantiated with a length (e.g., strings) or a length and an element type (e.g., sequences).

These categories are detailed in the subsequent clauses.

NOTE Within this building block, anonymous types, that is, the type resulting from an instantiation of a template type (see Building Block Anonymous Types) cannot be used directly. Instead, prior to any use, template types must be given a name through a **typedef** declaration. Therefore, as expressed in the following rules, referring to a simple type can be done either using directly its name, if it is a basic type, or using a scoped name, in all other cases:

```
(21)  <type_spec>          ::= <simple_type_spec>
(22)  <simple_type_spec>   ::= <base_type_spec>
                           |   <scoped_name>
```

7.4.1.4.4.2 Basic Types

Basic types are pre-existing types that represent numbers or characters. The set of basic types is defined by the following rules:

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

```
(23)  <base_type_spec>        ::= <integer_type>
                                  | <floating_pt_type>
                                  | <char_type>
                                  | <wide_char_type>
                                  | <boolean_type>
                                  | <octet_type>
(24)  <floating_pt_type>      ::= "float"
                                  | "double"
                                  | "long" "double"
(25)  <integer_type>          ::= <signed_int>
                                  | <unsigned_int>
(26)  <signed_int>            ::= <signed_short_int>
                                  | <signed_long_int>
                                  | <signed_longlong_int>
(27)  <signed_short_int>      ::= "short"
(28)  <signed_long_int>       ::= "long"
(29)  <signed_longlong_int>   ::= "long" "long"
(30)  <unsigned_int>          ::= <unsigned_short_int>
                                  | <unsigned_long_int>
                                  | <unsigned_longlong_int>
(31)  <unsigned_short_int>    ::= "unsigned" "short"
(32)  <unsigned_long_int>     ::= "unsigned" "long"
(33)  <unsigned_longlong_int> ::= "unsigned" "long" "long"
(34)  <char_type>             ::= "char"
(35)  <wide_char_type>        ::= "wchar"
(36)  <boolean_type>          ::= "boolean"
(37)  <octet_type>            ::= "octet"
```

Integer Types

IDL integer types are **short, unsigned short, long, unsigned long, long long**, and **unsigned long long** representing integer values in the range indicated below in Table 7.13.

Table 7.13 — Integer types

Integer type	Value range
N/A. See 7.4.1.3 Building Block Extended Data-Types	$-2^{7} \ldots 2^{7} - 1$
short	$-2^{15} \ldots 2^{15} - 1$
long	$-2^{31} \ldots 2^{31} - 1$
long long	$-2^{63} \ldots 2^{63} - 1$
N/A. See 7.4.1.3 Building Block Extended Data-Types	$0 \ldots 2^{8} - 1$
unsigned short	$0 \ldots 2^{16} - 1$
unsigned long	$0 \ldots 2^{32} - 1$
unsigned long long	$0 \ldots 2^{64} - 1$

Floating-Point Types

IDL floating-point types are **float, double**, and **long double**. The **float** type represents IEEE single-precision floating point numbers; the **double** type represents IEEE double-precision floating point numbers. The **long double** data type represents an IEEE double-extended floating-point number, which has an exponent of at least 15 bits in length and a

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

signed fraction of at least 64 bits. See IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, for a detailed specification.

Char Type

IDL defines a **char** data type that is an 8-bit quantity that (1) encodes a single-byte character from any byte-oriented code set, or (2) when used in an array, encodes a multi-byte character from a multi-byte code set.

Wide Char Type

IDL defines a **wchar** data type that encodes wide characters from any character set. The size of **wchar** is implementation-dependent.

Boolean Type

The **boolean** data type is used to denote a data item that can only take one of the values **TRUE** and **FALSE**.

Octet Type

The **octet** type is an opaque 8-bit quantity that is guaranteed not to undergo any change by the middleware.

7.4.1.4.4.3 Template Types

Template types are generic types that are parameterized by type of underlying elements and/or the number of elements. To be used as an actual type, such a generic definition must be *instantiated*, i.e., given parameter values, whose nature depends on the template type.

As specified in the following rule, template types are sequences (<**sequence_type**>), strings (<**string_type**>, wide strings (<**wide_string_type**>) and fixed-point numbers (<**fixed_pt_type**>).

```
(38)  <template_type_spec>   ::= <sequence_type>
                             |   <string_type>
                             |   <wide_string_type>
                             |   <fixed_pt_type>
```

Sequences

Sequences are defined according to the following syntax.

```
(39)  <sequence_type>        ::= "sequence" "<" <type_spec> "," <positive_int_const> ">"
                             |   "sequence" "<" <type_spec> ">"
```

As a template type, **sequence** has two parameters:

- The first non-optional parameter (<**type_spec**>) gives the type of each item in the sequence.
- The second optional parameter (<**positive_int_const**> is a positive integer constant that indicates the maximum size of the sequence. If it is given, the sequence is termed a *bounded* sequence. Otherwise the sequence is said *unbounded* and no maximum size is specified.

Before using a bounded or unbounded sequence, the length of the sequence must be set in a language-mapping dependent manner. If the bounded form is used, the length must be less than or equal to the maximum. Similarly after receiving a sequence, this value may be obtained in a language-mapping dependent manner.

Strings

IDL defines the string type string consisting of a list of all possible 8-bit quantities except null. A string is similar to a sequence of char. As with sequences of any type, prior to passing a string as a function argument (or as a field in a structure or union), the length of the string must be set in a language-mapping dependent manner. The syntax is:

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

(40) ***<string_type>*** ***::= "string" "<" <positive_int_const> ">"***
| "string"

The argument to the string declaration is the maximum size of the string (<**positive_int_const**>). If a positive integer maximum size is specified, the string is termed a bounded string. If no maximum size is specified, the string is termed an unbounded string. The actual length of a string is set at run-time and, if the bounded form is used, must be less than or equal to the maximum.

Strings are singled out as a separate type because many languages have special built-in functions or standard library functions for string manipulation. A separate string type may permit substantial optimization in the handling of strings compared to what can be done with sequences of general types.

Wstrings

The **wstring** data type represents a sequence of **wchar**, except the wide character null. The type **wstring** is similar to that of type **string**, except that its element type is **wchar** instead of **char**. The syntax for defining a **wstring** is:

(41) ***<wide_string_type>*** ***::= "wstring" "<" <positive_int_const> ">"***
| "wstring"

Fixed Type

The **fixed** data type represents a fixed-point decimal number of up to 31 significant digits. The syntax to declare a fixed data type is:

(42) ***<fixed_pt_type>*** ***::= "fixed" "<" <positive_int_const> "," <positive_int_const> ">"***

The first parameter is the number of total digits (up to 31), the second one the number of fractional digits, which must be less or equal to the former.

In case the fixed type specification is used in a constant declaration, those two parameters are omitted as they are automatically deduced from the constant value. The syntax is thus as follows:

(43) ***<fixed_pt_const_type>*** ***::= "fixed"***

NOTE The **fixed** data type will be mapped to the native fixed point capability of a programming language, if available. If there is not a native fixed point type, then the IDL mapping for that language will provide a fixed point data type. Applications that use the IDL fixed point type across multiple programming languages must take into account differences between the languages in handling rounding, overflow, and arithmetic precision.

7.4.1.4.4.4 Constructed Types

Constructed types are types that are created by an IDL specification.

As expressed in the following rule, structures (<**struct_dcl**>), unions (<**union_dcl**>) and enumerations (<**enum_dcl**>) are the constructed types supported in this building block:

(44) ***<constr_type_dcl>*** ***::= <struct_dcl>***
| <union_dcl>
| <enum_dcl>

All those constructs are presented in the following clauses.

Structures

A structure is a grouping of at least one member. The syntax to declare a structure is as follows:

(45) ***<struct_dcl>*** ***::= <struct_def>***
| <struct_forward_dcl>

(46) ***<struct_def>*** ***::= "struct" <identifier> "{" <member>+ "}"***

(47) ***<member>*** ***::= <type_spec> <declarators> ";"***

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

```
(67)  <declarators>        ::= <declarator> { "," <declarator> }*
(68)  <declarator>         ::= <simple_declarator>
```

A structure definition comprises:

- The **struct** keyword.
- The name given to the structure (<**identifier**>).
- The list of all structure members (<**member**>+) enclosed within braces ({}). Each member (<**member**>) is defined with a type specification (<**type_spec**>) followed by a list of at least one declarator (<**declarators**>). At least one member is required.

The name of a structure defines a new legal type that may be used anywhere such a type is legal in the grammar.

NOTE Members may be of any data type, including sequences or arrays. However, except when anonymous types are supported (cf. 7.4.14, for more details), sequences or arrays need to be given a name (with **typedef**) to be used in the member declaration.

Structures may be forward declared, in particular to allow the definition of recursive structures.

Unions

IDL unions are a cross between C unions and switch statements. They may host a value of one type to be chosen between several possible cases. IDL unions must be discriminated: they embed a discriminator that indicates which case is to be used for the current instance. The possible cases as well as the type of the discriminator are part of the union declaration, whose syntax is as follows:

```
(49)  <union_dcl>          ::= <union_def>
                           |   <union_forward_dcl>
(50)  <union_def>          ::= "union" <identifier> "switch" "(" <switch_type_spec> ")"
                               "{" <switch_body> "}"
(51)  <switch_type_spec>   ::= <integer_type>
                           |   <char_type>
                           |   <boolean_type>
                           |   <scoped_name>
(52)  <switch_body>        ::= <case>+
(53)  <case>               ::= <case_label>+ <element_spec> ";"
(54)  <case_label>         ::= "case" <const_expr> ":"
                           |   "default" ":"
(55)  <element_spec>       ::= <type_spec> <declarator>
```

A union declaration comprises:

- The **union** keyword.
- The name given to the union (<**identifier**>).
- The type for the discriminator (<**switch_type_spec**>). That type may be either one of the following types: **integer, char, boolean** or an enumeration, or a reference (<**scoped_name**>) to one of these.
- The list of all possible cases for the union (<**switch_body**>), enclosed within braces ({}). At least one case is required. Each possibility (<**case**>) comprises the form that the union takes (<**element_spec**>) when the discriminator takes the list of specified values (<**case_label**>+). Several case labels may be associated in a single case.

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

- A case label must be:
  - Either a constant expression (**<const_expr>**) matching (or automatically castable) to, the defined type of the discriminator.
  - Or the **default** keyword, to tag the case when the discriminator's value does not match the other possibilities. A default case can appear at most once.
- Each possible form for the union value is made of an existing IDL type (**<type_spec>**) followed by the name given to that form (**<declarator>**).

The name of a union defines a new legal type that may be used anywhere such a type is legal in the grammar.

It is not required that all possible values of the union discriminator be listed in the <**switch_body**>. The value of a union is the value of the discriminator together with one of the following:

1. If the discriminator value was explicitly listed in a case statement, the value of the element associated with that case statement.
2. If a default case label was specified, the value of the element associated with the default case label.
3. No additional value.

Access to the discriminator and to the related element is language-mapping dependent.

NOTE Name scoping rules require that the element declarators (all the <**declarator**> in the different <**element_spec**>) in a particular union be unique. If the <**switch_type_spec**> is an enumeration, the identifier for the enumeration is as well in the scope of the union; as a result, it must be distinct from the element declarators. The values of the constant expressions for the case labels of a single union definition must be distinct. A union type can contain a default label only where the values given in the non-default labels do not cover the entire range of the union's discriminant type.

NOTE While any ISO Latin-1 (8859-1) IDL character literal may be used in a <**case_label**> in a union definition whose discriminator type is char, not all of these characters are present in all code sets that may be used by implementation language compilers and/or runtimes, which may lead to some interoperability issue (typically leading to a **DATA_CONVERSION** system exception). Therefore, to ensure portability and interoperability, care must be exercised when assigning the <case_label> for a union member whose discriminator type is char. Due to this potential issue, use of char types as the discriminator type for unions is not recommended.

Enumerations

Enumerated types (enumerations) consist of ordered lists of enumerators. The syntax to create such a type is as follows:

(57) ***<enum_dcl>*** ***::= "enum" <identifier>***
"{" <enumerator> { "," <enumerator> } * "}"

(58) ***<enumerator>*** ***::= <identifier>***

An enumeration declaration comprises:

- The **enum** keyword.
- The name given to the enumeration (<**identifier**>).
- The list of the possible values (*enumerators*) that makes the enumeration, enclosed within braces ({}). Each enumerator is identified by a specific name (<**identifier**>). In case there are several enumerators, their names are separated by commas (,). An enumeration must contain at least one enumerator and no more than 2^{32}.

The name of an enumeration defines a new legal type that may be used anywhere such a type is legal in the grammar.

© ISO/IEC 2020 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020

NOTE The enumerated names must be mapped to a native data type capable of representing a maximally-sized enumeration. The order in which the identifiers are named in the specification of an enumeration defines the relative order of the identifiers. Any language mapping that permits two enumerators to be compared or defines successor/predecessor functions on enumerators must conform to this ordering relation.

Constructed Recursive Types and Forward Declarations

The IDL syntax allows the generation of recursive structures and unions via members that have a sequence type. For example, the following:

```
struct Foo;                                 // Forward declaration
typedef sequence<Foo> FooSeq;
struct Foo {
        long value;
        FooSeq chain;                       // Recursive
        };
```

The forward declaration for the structure enables the definition of the sequence type **FooSeq**, which is used as the type of the recursive member.

Forward declarations are legal for structures and unions. Their syntax is as follows:

(48) <struct_forward_dcl> ::= "struct" <identifier>

(56) <union_forward_dcl> ::= "union" <identifier>

A structure or union type is termed incomplete until its full definition is provided; that is, until the scope of the structure or union definition is closed by a terminating **}**;. For example:

```
struct Foo;             // Introduces Foo type name,
                        // Foo is incomplete now
// ...
struct Foo {
                        // ...
        };              // Foo is complete at this point
```

If a structure or union is forward declared, a definition of that structure or union must follow the forward declaration in the same compilation unit. If this rule is violated it shall be treated as an error. Multiple forward declarations of the same structure or union are legal.

If a sequence member of a structure or union refers to an incomplete type, the structure or union itself remains incomplete until the member's definition is completed. For example:

```
struct Foo;
typedef sequence<Foo> FooSeq;
struct Bar {
        long value;
        FooSeq chain;               // Use of incomplete type
        };                          // Bar itself remains incomplete
struct Foo {
        // ...
        };                          // Foo and Bar are complete
```

If this rule is violated, it shall be treated as an error.

Recursive definitions can span multiple levels. For example:

```
union Bar;                          //Forward declaration;
typedef sequence<Bar> BarSeq;
uniswitch (long) {                  //Define incomplete union
      case 0:
        long l_mem;
```

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 19516:2020