**ISO/IEC 14496-34:22##(X)**

ISO TC 1/SC 29/WG 03

Date: 2024-05-17

**Information technology — Coding of audio-visual objects — Part 34: Syntactic description language**

2nd Draft FDIS stage

**Warning for WDs and CDs**

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

© ISO 2022

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

Email: copyright@iso.org

Website: www.iso.org

Published in Switzerland

Contents

[Foreword v](#_Toc165642438)

[Introduction vi](#_Toc165642439)

[1 Scope 1](#_Toc165642440)

[2 Normative references 1](#_Toc165642441)

[3 Terms and definitions 1](#_Toc165642442)

[4 Notation 2](#_Toc165642443)

[4.1 Rule formatting 2](#_Toc165642444)

[4.2 Rule tokens 3](#_Toc165642445)

[4.3 Bitstream examples 3](#_Toc165642446)

[5 Language concepts and lexical elements 3](#_Toc165642447)

[5.1 Character set 3](#_Toc165642448)

[5.2 Whitespace 4](#_Toc165642449)

[5.3 Case sensitivity 4](#_Toc165642450)

[5.4 Comments 4](#_Toc165642451)

[5.5 Identifiers 4](#_Toc165642452)

[5.6 Punctuators 5](#_Toc165642453)

[5.7 Keywords 5](#_Toc165642455)

[5.8 Operators 6](#_Toc165642456)

[5.8.1 Operator precedence and associativity 7](#_Toc165642457)

[5.8.2 Specific operator behavior 7](#_Toc165642458)

[5.9 Expressions and evaluation 8](#_Toc165642459)

[5.10 Statements 9](#_Toc165642460)

[5.11 Built-in operators 9](#_Toc165642461)

[5.12 Numbers 10](#_Toc165642462)

[5.12.1 Endianness 10](#_Toc165642463)

[5.12.2 Representation of numbers 10](#_Toc165642464)

[5.12.3 Number limits 10](#_Toc165642465)

[5.12.4 Value coercion 11](#_Toc165642475)

[5.13 Logic values 11](#_Toc165642478)

[5.14 Binary literal values 11](#_Toc165642479)

[5.15 Hexadecimal literal values 11](#_Toc165642480)

[5.16 Integer, decimal and floating-point literal values 12](#_Toc165642481)

[5.17 String literal values 13](#_Toc165642482)

[5.18 Scope 14](#_Toc165642483)

[5.18.1 Parsable variable scope 15](#_Toc165642484)

[5.18.2 Non-parsable variable scope 15](#_Toc165642485)

[5.19 SDL specification limits 16](#_Toc165642486)

[6 Elementary data types 16](#_Toc165642487)

[6.1 Introduction 16](#_Toc165642488)

[6.2 Constant-length direct representation bit fields 17](#_Toc165642489)

[6.2.1 Alignment 17](#_Toc165642490)

[6.2.2 Type 17](#_Toc165642491)

[6.2.3 Length 17](#_Toc165642492)

[6.2.4 Value 18](#_Toc165642493)

[6.2.5 Constants 18](#_Toc165642494)

[6.2.6 Look-ahead 19](#_Toc165642495)

[6.3 Variable length direct representation bit fields 19](#_Toc165642497)

[6.4 Constant-length indirect representation bit fields 20](#_Toc165642498)

[6.5 Variable length indirect representation bit fields 22](#_Toc165642499)

[6.6 Variable length strings 23](#_Toc165642500)

[6.7 String value 24](#_Toc165642501)

[7 Composite data types 24](#_Toc165642502)

[7.1 Classes 24](#_Toc165642503)

[7.2 Base and derived classes 25](#_Toc165642504)

[7.3 Abstract classes 25](#_Toc165642505)

[7.4 Polymorphism in class declaration 26](#_Toc165642506)

[7.4.1 General 26](#_Toc165642507)

[7.4.2 Polymorphism for abstract classes 28](#_Toc165642508)

[7.5 Expandable classes 29](#_Toc165642510)

[7.6 Parameter types 30](#_Toc165642511)

[7.7 Arrays 34](#_Toc165642512)

[7.7.1 General 34](#_Toc165642513)

[7.7.2 Alignment 35](#_Toc165642514)

[7.8 Multi-dimensional arrays 36](#_Toc165642515)

[7.9 Partial arrays 36](#_Toc165642516)

[7.10 Implicit arrays 38](#_Toc165642517)

[8 Non-parsable variables 39](#_Toc165642518)

[8.1 General 39](#_Toc165642519)

[8.2 Elementary data types 40](#_Toc165642520)

[8.2.1 Value 40](#_Toc165642521)

[8.3 Arrays 40](#_Toc165642523)

[8.4 Multi-dimensional arrays 41](#_Toc165642524)

[9 Syntactic flow control 41](#_Toc165642525)

[9.1 Conditionals 41](#_Toc165642526)

[9.2 Loops 45](#_Toc165642527)

[Annex A (normative) SDL syntax 48](#_Toc165642528)

[Annex B (informative) SDL user guide 49](#_Toc165642529)

[B.1 Getting started 49](#_Toc165642530)

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

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](https://www.iso.org/directives-and-policies.html)).

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](https://www.iso.org/iso-standards-and-patents.html)).

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](https://www.iso.org/foreword-supplementary-information.html).

This document was prepared by Joint Technical Committee ISO/IEC 1, information technology, Subcommittee SC 29, coding of audio, picture, multimedia and hypermedia information.

A list of all parts in the ISO 14496 series can be found on the ISO website.

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](https://www.iso.org/members.html).

Introduction

This specification describes the mechanism with which bitstream syntax is documented in several standard parts such as ISO/IEC 14496 or ISO/IEC 23000. This mechanism is called the Syntactic Description Language (SDL) and is documented here in the form of syntactic and semantic rules.

The SDL builds on concepts defined in the C-like syntax used in ISO/IEC 11172–1:1993 and ISO/IEC 13818–1:2007 to define an extensible framework for describing bitstreams. This framework is inspired by class typing system concepts from the C++ and Java programming languages. SDL specialises this class typing system concept by providing facilities for defining bitstream-level quantities, and how they should be parsed.

The aim of this specification is to elevate the mechanism which has been used for over 20 years to specify encoded bitstreams as a standalone part. Initially defined as part of ISO/IEC 14496-1, this specification is backward-compatible compared to the SDL specified in ISO/IEC 14496-1. That is, a valid SDL specification based on ISO/IEC 14496-1 is also a valid specification based on this document. However, the inverse is not true.

Lexical elements of the SDL are described first, followed by elementary and composite type constructs to specify bitstreams. Finally, support for general purpose computation and syntactic control flow are addressed. Example SDL specification fragments and corresponding bitstreams are provided to clarify various concepts.

Information technology — Coding of audio-visual objects — Part 34: Syntactic description language

# Scope

This document specifies a syntactic description language for describing the structure of binary data. This document covers the representation of an SDL specification in plain text, the syntax of the SDL and the semantic rules of the SDL.

In scenarios where the usage or interpretation of the SDL may be ambiguous or undefined, this document attempts to specify whether such a scenario is considered an invalid specification or will result in undefined behavior.

NOTE: While the SDL borrows from and contains some aspects of a general-purpose programming language, it is not intended, nor is it suitable, to be used for such a purpose. This is reflected in the fact that many concepts related to general-purpose programming languages are not addressed in this document. Examples of concepts considered irrelevant to the SDL and therefore not addressed in this document include storage of an SDL specification in a file, compilation, execution, input/output, execution environment and machine architecture.

# Normative references

IETF RFC 4648, The Base16, Base32, and Base64 Data Encodings

IETF RFC 3629, UTF‑8, a transformation format of ISO 10646

IEEE Std 754-2019, IEEE Standard for Floating-Point Arithmetic

# Terms and definitions

For the purposes of this document, the following terms, abbreviations and definitions apply.

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

* ISO Online browsing platform: available at <https://www.iso.org/obp>
* IEC Electropedia: available at <https://www.electropedia.org/>

3.1

fixed length code

FLC

constant-length direct representation bit fields

3.2

syntactic description language

SDL

language defined by this specification that allows the description of a bitstream’s syntax

3.3  
composite type

a variable type in the SDL which consists of an aggregated structure of one or more elements e.g. array, class and map types

3.4  
declaration

a sequential unit of SDL tokens used to define the ‘shape’ of a variable value without causing data to be consumed from a bitstream or stored in the parsing context e.g. a class or map declaration

3.5  
definition

a sequential unit of SDL tokens used to create a parsable or non-parsable variable and its value, potentially causing data to be consumed from a bitstream or stored in the parsing context e.g. a parsable class variable definition or a non-parsable integer variable definition

3.6  
elementary type

individual numeric value types such as bit, integer and float types

3.7  
invalid specification

an SDL specification that is possible using the SDL syntax but is nonetheless considered invalid due to either semantic rules or the fact that such a specification would be impossible to interpret or process

3.8  
non-parsable variable

a variable whose value is only stored in the parsing context

3.9  
parsable variable

a variable whose value is initially only stored in a bitstream and once parsed the value is also stored in the parsing context.

3.10  
parsing context

an abstract concept of a volatile storage area for variable values. A computer program implementation would use computer memory for this and could choose to impose constraints such as storage size.

3.11  
specification

a formal description of an encoded binary bitstream using the SDL

3.12  
undefined behavior

behavior resulting in an undefined outcome based on the interpretation or processing of an SDL specification. Certain scenarios in this document are explicitly identified as resulting in undefined behavior. A computer program implementation or a standard presenting an SDL specification may choose to define expected behavior in these scenarios

3.13  
unspecified constraint

a constraint or limit on an SDL specification which would need to be defined for it to be viably processed in some manner by a computer program. As the scope of the SDL does not cover such processing, where appropriate such potential constraints are identified and explicitly left unspecified. A computer program implementation or a standard containing an SDL specification may choose to specify such constraints

# Notation

The body of this document outlines the concepts, lexical elements, syntax and semantics. Informal grammar rules are used throughout to introduce the syntax whilst a formal grammar is provided in Annex A.

## Rule formatting

The following formatting is used when presenting the informal grammar rules appearing in this document:

* Keywords, punctuators and operators are formatted using a bold monospace font e.g., **keyword**
* Constructs referenced across informal grammar rules are formatted using a bold, italic monospace font e.g., ***construct***
* Tokens used to express the rules (defined in 4.2) are formatted using non-italic, non-bold monospace font e.g., ‘[’ and ‘]’
* Identifiers and values are formatted using an italic variable width font e.g., *identifier*

## Rule tokens

The following tokens are used in the informal grammar rules appearing in this document:

* An optional element is indicated by surrounding it with ‘[’ and ‘]’ e.g., [*optional\_element*]
* Alternative elements are indicated by separating each alternative by ‘|’ e.g., *element\_1* | *element\_2*
* An unspecified sequence of one or more elements is indicated by ‘…’ e.g., *element*;…

## Bitstream examples

This document provides encoded bitstream examples demonstrating the behavior of various SDL constructs. The bitstream examples should be read left to right top to bottom. Line breaks are irrelevant and are used purely to allow annotation of the bitstream. Annotations appear after a comment marker ‘//’. For example the following indicates the 32-bit value 0xFF11 encoded in a binary bitstream:

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 // first two bytes

// blank lines are irrelevant

0 0 0 0 0 0 0 1 // third byte

0 0 0 0 // another 4 bits

0 0 0 1 // last 4 bits

Examples always have a starting parse position. If parsing has just started, the example's starting parse position will be before the first bit in the bitstream. If parsing has already commenced, the example’s starting parse position will be after the last parsed bit. In both cases the example's starting parse position is before the first bit that is relevant to the example. To emphasize the fact that the example may be part of a larger bitstream, examples will always be preceded and succeeded by ‘…’. As an example:

...

0 0 0 0 // first 4 bits

0 0 0 1 // last 4 bits

...

Where skipping an indeterminate number of bits to achieve alignment is part of the example it is indicated as follows:

...

<skip 0..7 bits> // aligned(8)

0 0 0 0 // first 4 bits

0 0 0 1 // last 4 bits

...

# Language concepts and lexical elements

## Character set

The basic character set used to write an SDL specification consists of:

The 26 uppercase letters of the Latin alphabet:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The 26 lowercase letters of the Latin alphabet:

a b c d e f g h i j k l m n o p q r s t u v w x y z

The 10 decimal digits:

0 1 2 3 4 5 6 7 8 9

The following 29 graphic characters:

! " # % & ' ( ) \* + , - . / : ; < = > ? [ \ ] ^ \_ { | } ~

Additionally, in certain string literal values, an SDL specification may contain multibyte characters.

NOTE String literal values are defined in subclause 5.17

The character set and encoding used to store an SDL specification in a file is not specified in this document. A computer program implementation or a standard containing an SDL specification may choose to specify such details.

## Whitespace

One or more space, tab or carriage return characters constitute a whitespace and serve to demarcate tokens in an SDL specification. Whitespace is required wherever parsing of tokens in an SDL specification would be ambiguous. For example, the following are all valid and equivalent:

EXAMPLE 1 ⎯

int ( 10 ) i = -1 .. 3;

int(10)i=-1..3;

int( 10) i= - 1 ..3;

For example, the following is an invalid specification as tokenization of unsigned int is not possible:

EXAMPLE 2 ⎯

unsignedint i; // missing whitespace causing token ambiguity

Indentation using whitespace is used only for assisting in the readability of an SDL specification and has no effect on its syntactic or semantic meaning.

## Case sensitivity

The SDL is case sensitive.

## Comments

Comments starts by // and ends at the end of the current line. A comment may be preceded by any text between the start of the line and the start of the comment.

Rule S.1: Comments

**//** This is a comment.

## Identifiers

Identifiers can be comprised of a mixture of upper- and lower-case Latin alphabetic characters, digits and underscore (e.g. ‘\_’). An identifier shall include at least one alphabetic character. For example, the following are all valid identifiers:

EXAMPLE 1 ⎯

myVar

My\_Var

My2ndVar

2D\_Region[2]

\_2d\_region

It is illegal to define an identifier which conflicts (ignoring case) with SDL syntax items such as keywords, binary, hexadecimal and string literal prefixes. For example, the following identifiers are all invalid specifications:

EXAMPLE 2 ⎯

u // conflict with UTF-16 literal string prefix

0b // conflict with binary literal value prefix

0B // case-insensitive conflict with binary literal value prefix

Map // case-insensitive conflict with keyword "map"

1e2 // conflict with floating point literal value

2\_2 // identifier shall contain at least one alphabetic character

As the SDL is case sensitive it is possible but not recommended to define an identifier which has the same letters but is distinguished only via case from a keyword. For example:

Break // valid but not recommended identifier as it may be confused with the keyword break

NOTE: When defining identifiers in an SDL it is recommended to consider that automated tools may use these identifiers directly in a diverse range of programming languages. It is therefore recommended to avoid defining identifiers which start with digits or underscores as these identifiers may be illegal or have specific meaning in some programming languages. It is also recommended to avoid identifiers which may conflict with keywords in programming languages e.g. public.

## Punctuators

The following punctuator tokens are defined in the SDL:

**(** open parenthesis

**)** close parenthesis

**{** open brace

**}** close brace

**[** open bracket

**]** close bracket

**:** colon

**;** semicolon

**,** comma

**"** double quote

## Keywords

The following keyword tokens are defined in the SDL:

**abstract**

**aligned**

**base64string**

**bit**

**break**

**case**

**class**

**const**

**default**

**do**

**double**

**else**

**expandable**

**extends**

**float**

**for**

**if**

**int**

**lengthof**

**map**

**switch**

**unsigned**

**utf8string**

**utf8list**

**utfstring**

**while**

NOTE double is a legacy alias for the keyword float. The use of the keyword double is no longer recommended.

## Operators

The following arithmetic, logical and language specific operators are defined in the SDL:

**.** class member access e.g. *a***.***b* (discussed further in 5.11)

**[]**  array element access e.g. *a***[***b***]**

**++** postfix increment e.g. *a***++**

**--** postfix decrement e.g. *a***--**

**+** (unary) plus e.g. **+***a*

**-** (unary) negation e.g. **-***a*

**\*** multiplication

**/** division

**%** modulus

**+** addition

**-** subtraction

**<<** bitwise shift left

**>>** bitwise shift right

**<** relational less than

**<=** relational less than or equal

**>** relational greater than

**>=** relational greater than or equal

**==** relational equal

**!=** relational not equal

**&** bitwise AND

**|** bitwise OR

**&&** logical AND

**||** logical OR

**=** assignment (discussed further in 5.9)

### Operator precedence and associativity

The operators are presented below in order of descending precedence. Operators appearing on an earlier line have higher precedence. Operators appearing on the same line have equal precedence. In general, operators with equal precedence have left-to-right associativity. However some operators have right-to-left associativity and these are indicated below:

**[] .**

**++ -- +**(unary) **-**(unary) (right-to-left associativity)

**\* / %**

**+ -**

**<< >>**

**< <= > >=**

**== !=**

**&**

**|**

**&&**

**||**

**=**  (right-to-left associativity)

A pair of matching parentheses (e.g. ‘**(**‘ and ‘**)**’) can be used to group operations to apply specific precedence and associativity which differ from the default. Parentheses may be used to group more than one operator. Parentheses may be nested. For example:

EXAMPLE ⎯

7 \* 2 - 4 + 2 // equals 12

7 \* (2 – (4 + 2)) // equals -28

### Specific operator behavior

The array element access operator (i.e. ‘**[]**’) cannot be used with float value operands or negative integer values. The behavior of the array element access operator when the operand specifies an element index equal to or greater than the size of the array is not defined by the SDL which may lead to undefined behavior.

The modulus operator (i.e. ‘**%**’) cannot be used with float value operands. The sign of the result for the modulus operator with negative operands is not defined by the SDL which may lead to undefined behavior. The result of the modulus operator where the value of the second operand is zero is not defined by the SDL which may lead to undefined behavior.

The division operator (i.e. ‘**/**’) applied to integer values truncates the result meaning any fractional part of the value discarded. The direction of truncation for integer division with negative operands is not defined by the SDL which may lead to undefined behavior. The value of division where the value of the second operand is zero is not defined by the SDL which may lead to undefined behavior.

The left shift operator (i.e. ‘**<<**’) fills vacated bits with zero.

The right shift operator (i.e. ‘**>>**’) applied to an unsigned int value fills vacated bits with zero. The behaviour of the right shift operator applied to a signed int value is not defined by the SDL which may lead to undefined behavior.

Usage of the equality (i.e. ‘**==**’) and inequality operators (i.e. ‘**<**’, ‘**<=**’, '**>**’, ‘**>=**’ and ‘**!=**’) with operands of differing types is not defined by the SDL which may lead to undefined behavior. A computer program implementation or a standard containing an SDL specification may choose to specify such details.

## Expressions and evaluation

An expression is a sequence of operators and operands which are evaluated to produce a result which is either a value or a reference to a variable. The values of the operands are evaluated first before evaluating the result of the operator.

The assignment operator may only appear once in an expression. For example the following is an invalid specification:

EXAMPLE 1 ⎯

foo1 = foo2 = 1; // illegal usage of assignment more than once in an expression

For the assignment operator (i.e. **=**), the evaluation order of operands is left to right. For example:

EXAMPLE 2 ⎯

i = 0;

a[i] = i++; // i is used as an index in the array before it is incremented

// note that i is used as the right-hand value before the postfix increment

// so after this expression: a[0] == 0 and i == 1.

For the postfix operators (i.e. ‘**++**’ and ‘**--**‘), the value of the operand is used in the expression before the result of the operand is evaluated. For example:

EXAMPLE 3 ⎯

i = 1;

j = i++; // j now equals 1 and i equals 2

For the logical operators (i.e. **&&** and **||**), the evaluation order of operands is left-to-right and evaluation will stop as soon as the logical result value is determined. For example:

EXAMPLE 4 ⎯

i = 0;

if (1 || i++) { // i++ is not evaluates as 1 is equivalent to logical true

j = i; // j will be equal to 0

}

For other binary operators, the evaluation order of the two operands is not defined by the SDL which may lead to undefined behavior. A computer program implementation or a standard presenting an SDL specification may choose to define expected behavior in these scenarios.

In some scenarios, although the evaluation order is not deterministic, the result may always be the same. Such cases, although valid, should be avoided. For example:

EXAMPLE 5 ⎯

x = 1;

y = x++ + x++; // regardless of evaluation order, x will be used once before a postfix

// and once after a postfix increment i.e. y = 1 + 2 or y = 2 + 1

// both scenarios result in y == 3 and x == 3

## Statements

A statement consists of an expression followed by a semicolon. By surrounding a number of statements with a pair of matching braces (**{** and **}**), the statements are grouped into a block which is itself treated as a single statement. Such a block is not followed by a semicolon.

Statements are evaluated in sequential order except for scenarios involving syntactic flow control (defined in 9).

## Built-in operators

This **lengthof()** operator returns the length, in bits, of the variable contained in parentheses. The length is the number of bits that was most recently used to parse the variable value. A return value of 0 means that no bits were parsed for this variable e.g. all parsable variable members of a class were contained in conditional clauses whose conditions were not met. It is illegal to use the **lengthof()** operator with a non-parsable variable.

EXAMPLE 1 ⎯

int(3) foo; // lengthof(foo) == 3

class A { // classes are defined later in this document

int (5) b; // parsable variable

int c; // non-parsable variable

}

A a; // lengthof(a) == 5

class Point { // classes are defined later on in this document

int x;

int y;

}

map points (Point) { // maps are defined later on in this document

0b00, {100, 100},

0b01, {200, 200}

}

Point(points) p; // lengthof(p) == 2

utf8string myString; // if myString == "hello" then lengthof(myString) == 48

// i.e. 5 characters + null termination

Alignment attributes of parsable variables do not affect the value returned by the **lengthof()** operator as any skipped bits in the bitstream are not used to store the value of the variable.

EXAMPLE 2 ⎯

int (3) foo1; // lengthof(foo1) == 3

aligned(16) int (3) foo2; // lengthof(foo2) == 3

Rule O.1: lengthof() Operator

lengthof(variable)

The range operator defines the inclusive range of numbers from the specified *min\_value* up to the specified *max\_value*. The *min\_value* and *max\_value* shall be of the same type and the *max\_value* shall be greater than or equal to the *min\_value*.

Rule O.2: Range operator

min\_value ..max\_value

The following are all valid and unique range expressions:

EXAMPLE 1 ⎯

-1..3

0b01..0b11

MinID..MaxID

0x01 .. MaxID

-1e5 .. 1e5

A period (i.e., ‘**.**’) can be used to access members of a **class** variable.

Rule O.3: Class member access operator

class\_variable\_identifier.member\_identifier…

The operator may be repeated if the member is itself a class variable e.g.:

EXAMPLE 2 ⎯

level1.level2.level3

## Numbers

### Endianness

All number values shall be represented in the bitstream with the most significant byte first, and the most significant bit first.

For the purposes of expression evaluation, the storage of number values in the parsing context (for both parsable and non-parsable variables) is also assumed to be with the most significant byte first, and the most significant bit first.

### Representation of numbers

The length of an unsigned or signed integer value is the number of bits used to represent the value in the bitstream or in the parsing context. For a signed integer one bit of the length is used to represent the sign. Signed integer values shall be represented using the two’s-complement representation.

The values of float variables shall be represented in the bitstream or in the parsing context using the interchange format for binary floating-point numbers defined in IEEE Std 754-2019 for the corresponding defined bit length.

Although the SDL supports signed zero for floating point literals, the interpretation and use of such values is unspecified behavior. Scenarios in which negative zero values are generated and how they are coerced is also unspecified behavior.

### Number limits

Parsable variables have defined width attributes which explicitly define the maximum and minimum (for signed numbers) limits of values which can be found in encoded bitstreams.

Non-parsable variables values (which are stored only in the parsing context) and parsable variable values once parsed from the bitstream (which are then stored in the parsing context) do not have constraints on the storage width and therefore do not have number limits explicitly defined by the SDL. Care should be taken when creating SDL specifications to avoid unspecified behavior. As an example:

EXAMPLE ⎯

bit b = 0;

b++; // b is now equal to 1

b++; // undefined as to whether this causes an error or b wraps to 0 or increments to 2

A computer program implementation or a standard containing an SDL specification may choose to specify such details.

### Value coercion

Apart from logic value coercion (defined in 5.13) the SDL does not specify behaviour with respect to coercion of values. Therefore conversion between values of different types or values using different widths or ranges of the same type will result in unspecified behaviour. Care should be taken when creating SDL specifications to avoid unspecified behavior. As an example:

EXAMPLE ⎯

int(3) foo = 15; // undefined as to whether the effective value will be truncated to 7

int i = 4;

float f = 2 \* i; // undefined coercion from int to float values

unsigned int u\_max = MAX\_UNSIGNED\_INT;

int max = u\_max; // undefined conversion between signed and unsigned number limits

## Logic values

Although SDL does not define a Boolean type, the concept of logical values of true and false are required when evaluating expressions involving logical operators and when evaluating conditions used within syntactic flow control statements. To accommodate this, a numeric value of zero is coerced to a logic value of false when required and a non-zero value is coerced to a logic value of true.

## Binary literal values

To designate literal binary values, the **0b** prefix shall be used followed by a series of one or more 0 and 1 characters.

**Rule S.2: Binary literal value**

0bbinary\_value

A period (‘.’) may be optionally placed after every four digits for readability.

EXAMPLE 1 ⎯

0b00100101

0b0010.0101

As the SDL is case sensitive, usage of a capital ‘**B**’ in the binary literal prefix is invalid. For example the following is an invalid specification:

EXAMPLE 2 ⎯

0B00100101 // not a binary literal as uppercase B has been used

## Hexadecimal literal values

To designate literal hexadecimal values, the prefix 0x shall be used followed by a series of characters in the range A to F and digits in the range 0 to 9.

Rule S.3: Hexadecimal literal value

0xhexadecimal\_value

A period (i.e., ‘**.**’) may be optionally placed after every four characters for readability:

EXAMPLE 1 ⎯

0xCAFEBEEF

0xCAFE.BEEF

As the SDL is case sensitive, usage of a capital ‘**X**’ in the hexadecimal literal prefix is invalid. For example the following is an invalid specification:

EXAMPLE 2 ⎯

0XCAFEBEEF // not a hexadecimal literal as uppercase X has been used

0xCaFE // invalid hexadecimal literal as lowercase a has been used

NOTE 0x0F is equivalent to 0b0000.1111.

## Integer, decimal and floating-point literal values

Integer literal values can be defined using digits and an optional signed unary operator (i.e. ‘**+**’, ‘**-**'). The following are valid integer literal values:

EXAMPLE 1 ⎯

200

-200

0

The following integer literal values are invalid specifications:

EXAMPLE 2 ⎯

-0 // signed zero is illegal

+0 // signed zero is illegal

002 // leading zeroes are illegal

Decimal literal values can be defined using digits, an optional signed unary operator (i.e. ‘**+**’, ‘**-**') and an optional decimal separator (i.e. ‘**.**’). The following are valid decimal values:

EXAMPLE 3 ⎯

58

58.0

58.3

0.4

+2.3

-2.3000

The following decimal literal values are invalid specifications:

EXAMPLE 4 ⎯

-0.0 // signed zero is illegal

002.3 // leading zeroes are illegal

Floating-point literal values can be defined using digits, an optional signed unary operator (i.e. ‘**+**’, ‘**-**'), an optional decimal separator (i.e. ‘**.**’) and an optional exponent indicator (i.e. ‘**e**’). The following are valid floating-point values:

EXAMPLE 5 ⎯

58

-58.0

123e67

123.456e-67

0.4e-67

-0 // negative zero

+0 // positive zero

The following floating-point literal values are invalid specifications:

EXAMPLE 6 ⎯

123E67 // uppercase E is illegal

00123e67 // leading zeroes are illegal

123e067 // leading zeroes are illegal

## String literal values

String literal values can be defined as a sequence of characters enclosed in double quotation marks (e.g. **"** and **"**) with an optional encoding prefix. The encoding prefix is one of the following: u8 for UTF-8 string literal or ufor UTF-16 string literal. Lack of an encoding prefix indicates usage of the basic SDL character set.

Literal string values may contain characters from the basic SDL character set, UTF-8 encoded multi-byte character for UTF-8 string values and UTF-16 encoded multi-byte character for UTF-16 string values. Literal string values may contain the space character. Literal string values shall not contain other white space characters such as tab or carriage return.

The following are valid string literal values:

EXAMPLE 1 ⎯

"hello world " // basic character set

u8"hello world §" // UTF-8 encoded characters

u"hello world §" // UTF-16 encoded characters

The following are invalid specifications:

EXAMPLE 2 ⎯

"hello // illegal carriage return character within...

world" // ... string literal value

"hello world §" // illegal UTF-8 encoded characters in non UTF-8 string literal

To allow the usage of the double quote character (i.e. ‘**"**’) as a literal string value within a pair of double quotation marks (which are used to demarcate the literal string) the escape sequence **\"** is defined to mean the character **"**. For example:

EXAMPLE 3 ⎯

"It is \"good\"" // to be interpreted as: It is "good"

To aid in visual formatting of an SDL specification, adjacent string literal tokens should be considered as concatenated values. For example, the following are equivalent:

EXAMPLE 4 ⎯

"hello world"

"hello" " world"

String literal tokens on sequential lines should also be considered concatenated. For example, the following is equivalent to the previous examples:

EXAMPLE 5 ⎯

"hello"

" world"

It is illegal to place non-compatible string literal tokens adjacent to each other. For example these are invalid specifications:

EXAMPLE 6 ⎯

u8"hello" u"world" // illegal concatenation of UTF-8 and UTF-16 values

u8"hello" "world" // illegal concatenation of UTF-8 and basic character values

Universal character names may be used in string literals to designate characters that are not in the basic character set or may not be printable. A universal character name can be defined using 4 digits prefixed with **\u** or 8 digits prefixed with **\U** (as specified by ISO/IEC 10646). The following are valid string literal values:

EXAMPLE 7 ⎯

u8"\u1234" // to be interpreted as: ሴ

u8"\U00001234" // to be interpreted as: ሴ

To accommodate the usage of the character sequences **\u** and **\U** for universal character names and **\"** for double quote, the escape sequence **\\** is also defined to mean the character **\**. For example:

EXAMPLE 8 ⎯

u8"\\u1234" // to be interpreted as: \u1234

u8"\\\\" // to be interpreted as: \\

"\\\"" // to be interpreted as: \"

Note that further escape character sequences such as **\n** and **\t** (which other computer languages may ascribe special meaning to) are legal within string literal values but have no special meaning. A computer program implementation or a standard containing an SDL specification may choose to ascribe such meanings to such character sequences.

## Scope

An identifier can denote a variable, a member of a map, a member of a class or a class parameter. The same identifier can denote different entities at different points in an SDL specification.

The scope of an identifier is the part of an SDL specification within which the identifier is visible and can be referenced.

An identifier can only be defined once within a single scope. If an identifier defined in an outer scope is defined again within a nested inner scope, then within the inner scope, references to the identifier will refer to the inner scope variable definition and the outer scope variable definition is not visible.

There are three kinds of scope: global, class and block.

If the declaration of an identifier appears outside of any block or parameter list, the identifier has global scope, which terminates at the end of the SDL specification. The scope of a global class declaration lasts from the point at which it is declared to the end of the SDL specification.

[Editor’s note: The above implies no global constants or maps.]

[Editor’s note: The above implies no forward references are therefore no recursive/self-referential declarations.]

[Editor’s note: How is the entry point specifier or determined? Is it specified outside of the SDL specification?]

[Editor’s note: Recent proposal to address the previous notes was to state:

The only elements allowed in global scope are class declarations and class definitions.

NOTE If one SDL specification introduces global scope elements in separate sections of a document or makes use of global scope elements declared in another SDL specification then that specification may need to define details on the interpretation of element ordering.]

If the identifier appears within the list of parameters for a class declaration or within the top-level block declaring the class, the identifier has class scope. It is accessible anywhere within the block which declares the class. Class scope variables can be accessed outside of the class using the class member access operator (i.e. ‘**.**’).

An identifier declared within a block (introduced by the character '**{**' and exited by a matching paired character '**}**' has block scope. It is accessible from the point it is declared until the end of the block.

The SDL defines two specific scope behaviors for parsable and non-parsable variables which differ from other well-known programming languages. These are presented in the following two clauses.

### Parsable variable scope

Parsable variables (defined in 6 and 7) defined within a class declaration, regardless of nested block scopes or conditional branch (defined in 9.1) scopes have class scope, i.e., they are available as class member variables.

NOTE This scoping behaviour differs from other well-known programming languages.

For example:

EXAMPLE ⎯

class A {

// class scope starts...

// foo1 is not yet defined

int (5) foo1; // parsable variable

// foo1 is now defined and accessible in class scope

{

// nested block scope starts...

// foo1 can be referenced here due to both class scope and outer block scope

// foo2 is not yet defined

int (5) foo2; // parsable variable

// foo2 is now defined and accessible in block scope AND class scope

}

// foo2 can be referenced here as it has class scope

if (foo1 == 0) {

// conditional block scope starts...

int (5) foo3; // parsable variable

// foo3 is now defined and accessible in block scope AND class scope

}

// foo3 can be referenced here as it has class scope

}

A a;

// a.foo1 is accessible

// a.foo2 is accessible

// a.foo3 is accessible

### Non-parsable variable scope

In general, non-parsable variables (defined in 8), have block scope. The exception to this is non-parsable variables defined in the top-level scope of a class which have class scope i.e. they are available as class member variables.

For example:

EXAMPLE ⎯

class B {

// class scope starts...

// bar1 is not yet defined

int bar1; // non-parsable variable

// bar1 is now defined and accessible in class scope

{

// nested block scope starts...

// bar1 can be referenced here due to both class scope and outer block scope

// bar2 is not yet defined

int bar2; // non-parsable variable

// bar2 is now defined and only accessible in block scope

}

if (bar1 == 0) {

// conditional block scope starts...

int bar3; // non-parsable variable

// bar3 is now defined and only accessible in block scope

}

}

B b;

// b.bar1 is accessible

// b.bar2 is NOT accessible

// b.bar3 is NOT accessible

## SDL specification limits

As this document does not cover parsing or translation of SDL specifications, the following are explicitly identified as unspecified constraints on an SDL specification:

* Maximum levels of block nesting
* Maximum levels of expression nesting
* Maximum levels of class nesting
* Maximum levels of map nesting
* Maximum number of identifiers
* Maximum number of class members
* Maximum number of labelled statements in a switch clause
* Maximum number of characters in a string literal
* Maximum number of parameters in a class parameter list
* Maximum character length of a string literal value
* Maximum number of concatenated adjacent string literal values
* Maximum number of dimensions in a multi-dimensional array
* Maximum number of elements in each dimension of an array

# Elementary data types

## Introduction

The SDL defines the following elementary data types as listed below and then described in more detail in the subsequent subclauses:

1. **Constant-length direct representation bit fields or Fixed Length Codes (FLCs)**. These describe the encoded value exactly as it is to be used by the appropriate decoding process. (subclause 6.2)
2. **Variable length direct representation bit fields, or parametric FLCs**. These are FLCs for which the actual length is determined by the context of the bitstream (e.g., the value of another parameter). (subclause 6.3)
3. **Constant-length indirect representation bit fields**. These require an extra lookup using an encoded value FLC into an appropriate table or variable to obtain the desired value or set of values. (subclause 6.4)
4. **Variable-length indirect representation bit fields (e.g., Huffman codes)**. These require an extra lookup using an encoded value parametric FLC into an appropriate table or variable to obtain the desired valu**e** or set of values. (subclause 6.5)
5. **Variable length strings**. These represent a series of characters. (subclause 6.6)

## Constant-length direct representation bit fields

Constant-length direct representation bit fields, or FLCs, shall be represented as:

Rule E.1: Elementary data types

[const] [aligned[(modifier)]] type(length) identifier [**=** value]**;**

### Alignment

The keyword aligned indicates that the data is aligned on a byte boundary.

For example, a 16‑bit value aligned on a byte boundary:

EXAMPLE 1 ⎯

aligned bit(16) foo;

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

<skip 0..7 bits> // byte aligned

0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0 // foo = 0x1248

...

An optional numeric modifier attribute may be used to signify alignment on boundaries other than byte. Allowed values are 8, 16, 32, 64, and 128. Any skipped bits due to alignment shall have the value ‘0’.

For example, an 8‑bit value aligned on a 2‑byte boundary:

EXAMPLE 3 ⎯

aligned(16) bit(8) foo;

An example bitstream for this would be:

EXAMPLE 4 ⎯

...

<skip 0..15 bits> // 2-byte aligned

0 0 0 1 0 0 1 0 // foo = 0x12

...

### Type

The grammar construct type may be any of the following: int for signed integer, unsigned int for unsigned integer, float for floating point, and bit for raw binary data.

NOTE double is a legacy alias for the keyword float. The use of the keyword double is no longer recommended.

### Length

The length attribute indicates that the length of the element value in bits, as it is stored in the bitstream. When the grammar construct type is float, then the length attribute shall be equal to 16, 32, 64, 128 or 256.

NOTE As some of those bit depths may be uncommon for floats, it is recommended to verify that the expected environment implementing a given specification supports the chosen bit depth.

Refer to 5.12.3 for a discussion on the maximum value of *length*.

For example, a 5‑bit unsigned integer that is parsable would be represented as:

EXAMPLE 1 ⎯

unsigned int(5) parsable\_variable;

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

1 0 0 1 0 // parsable\_variable = 0x12

...

### Value

The value attribute shall be present only when the value is constrained to a single value (e.g., start codes) or a range of values (i.e., ‘0x01..0xAF’).

A variable may be used when defining a single value or range of values for the *value* attribute. For example:

EXAMPLE 1 ⎯

unsigned int(8) min = 0..4;

unsigned int(8) max = 4..7;

unsigned int(8) foo = min..max;

The single value or range of values defined shall be valid for the defined variable type. For example the following is an invalid specification:

EXAMPLE 2 ⎯

unsigned int(8) foo = 0..5e10; // illegal usage of float value

Refer to 5.12.3 for a discussion on the limits of *value*.

Refer to 5.12.4 for a discussion on the coercion of *value*.

Parsable elementary data type variables with definitions that fall outside the flow of parsing shall have a default value of 0.

### Constants

Constants shall be defined using the keyword const.

EXAMPLE 1 ⎯

const int(8) SOME\_VALUE; // the parsed value will be immutable

const bit(2) BIT\_PATTERN=0b01; // the parsed value shall be 0b01 and will be immutable

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

0 0 0 1 0 0 1 0 // SOME\_VALUE = 0x12 and is now immutable

0 1 // BIT\_PATTERN = 0x1 (error if bitstream not "0 1") and is now immutable

...

A **const** variable value shall not be modified. For example, the following is an invalid specification:

EXAMPLE 3 ⎯

const int(8) PRECISION = 10;

precision++; // mutating a constant value is illegal

### Look-ahead

In some scenarios, it may be desirable to examine the immediately following bits in the bitstream, without consuming these bits. To support this behavior, a ‘\*’ character shall be placed after the parse *length* parentheses to modify the parse size semantics.

Rule E.2: Look-ahead parsing

[aligned[(modifier)]] type (length)\* identifier [**=** value];

The keyword aligned and its modifier have the same meaning as in subclause 6.2.1.

For example, the value of the next 32 bits in the bitstream can be read as an unsigned integer without advancing the current position in the bitstream using the following representation:

EXAMPLE 1 ⎯

unsigned int(8)\* next\_byte;

// check if high-bit set

if (next\_byte & 0x80) {

// if so parse as a 16-bit value

unsigned int(16) value;

}

else {

// if not parse as an 8-bit value

unsigned int(8) value;

}

NOTE The conditional **if**-then-**else** construct is defined in subclause 9.1.

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

// next\_byte = 0x87

1 0 0 0 0 1 1 1 0 1 0 0 1 0 1 0 // value = 0x8746

...

## Variable length direct representation bit fields

The case of variable length direct representation bit fields, or parametric FLCs is covered by Rule E.1, by allowing the length attribute to be a variable included in the bitstream, a non-parsable variable, or an expression involving such variables. In case the length attribute is determined by an expression, the result of this expression shall be a strictly positive integer value.

For example, in the following representation, the size of the variable DC is determined by the 3‑bit unsigned integer value precision:

EXAMPLE 1 ⎯

unsigned int(3) precision;

int(precision) DC;

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

1 0 1 // precision = 5

1 0 0 1 1 // DC = -3

...

## Constant-length indirect representation bit fields

Indirect representation indicates that the actual value of the element at hand is indirectly specified by the bitstream using a table or map structure. In other words, the value extracted from the bitstream, which is an FLC, is an index to a map from which the final desired value is extracted. This indirection may be expressed by defining the map itself:

Rule E.3: Maps

map map\_identifier (output\_type) {

index**,** {value[**,** …] }[,

…]

}

The content of the map is declared as a set of pairs of input *index* values and output *values*. The *index* values shall be unique.

NOTE The syntax for maps does not allow to specify the alignment of map index values. Achieving such alignment is the responsibility of the SDL specification author.

The input type of a map (the index specified in the first column) shall always be bit and therefore *index* values always expressed as binary literal values. For the case of constant-length indirect representation bit fields, these binary literal values shall all be the same length. The length in bits of the index is the number of bits read from the bitstream.

The output\_type of a map shall be either a ***type*** or a declared **class** (classes are defined in 7.1).

Output values used to populate the output\_type shall be specified as aggregates surrounded by curly braces (“{“ and “}”). Output values shall only be literal numeric values or constants (with the exception of parsed values as defined in 4).

For example, a map that relates an input binary value to an output populated YUVblocks **class** (classes are defined in subclause 7.1) can be declared using the following representation:

EXAMPLE 1 ⎯

class YUVblocks {// classes are defined later on in this document

int Yblocks;

int Ublocks;

int Vblocks;

}

// a table that relates the chroma format with the number of blocks

// per signal component

map blocks\_per\_component (YUVblocks) {

0b00, {4, 1, 1},

0b01, {4, 2, 2},

0b10, {4, 4, 4}

}

In the above example the class YUVblocks contains only non-parsable member variables (defined in 8). Although the SDL supports using a declared **class** containing parsable member variables as the *output\_type* of a map, such usage will result in unspecified behavior.

As another example, a map that relates an input binary value to an output populated **int** value can be declared using the following representation:

EXAMPLE 2 ⎯

map offsets (int) {

0b00, {1024},

0b01, {2048},

0b10, {4096}

}

As class declarations may include nested hierarchies, the aggregate output values may also include nesting. As an example:

EXAMPLE 3 ⎯

class Foo {

int f1;

f2;

}

class Bar {

int b1;

Foo foo1;

}

map barMap (Bar) {

0b00, {0, { 1, 1} },

0b01, {4, { 2, 2} },

0b10, {4, { 4, 4} }

}

The type and number of constituent *values* within the aggregate output values of the map declaration shall match the corresponding constituent type within the map’s *output\_type*. For example, the following is an invalid specification:

EXAMPLE 4 ⎯

map illegal (int) {

0b00, {1.1}, // illegal float value for an integer output

0b01, {2, 2}, // illegal multiple values for a single value output

0b10, {} // illegal missing value

}

The next rule describes the usage of map declarations to define variables.

Rule E.4: Mapped data types

output\_type(map\_identifier) *map\_variable\_*identifier**;**

The output\_type of the variable shall be identical to the output\_type declared for the map.

For example, the following makes use of the blocks\_per\_component **map** declared above which has a YUVBlocks **class** output type:

EXAMPLE 4 ⎯

YUVblocks(blocks\_per\_component) chroma\_format;

if (chroma\_format.Ublocks != 4) {

unsigned int(8) u\_width;

unsigned int(8) u\_height;

}

An example bitstream for this would be:

EXAMPLE 5 ⎯

...

0 1 // input 0b01 maps to

// chroma\_format = { Yblocks = 4, Ublocks = 2, Vblocks = 2 }

0 0 0 1 0 0 0 0 // u\_width = 16

0 0 0 1 0 0 0 0 // u\_height = 16

...

As another example, the following makes use of the offsets **map** declared above which has an **int** output type:

EXAMPLE 6 ⎯

int(offsets) index\_offset;

if (index\_offset == 2048) {

unsigned int(6) foo;

}

An example bitstream for this would be:

EXAMPLE 7 ⎯

...

0 1 // input 0b01 maps to

// index\_offset = 2048

0 1 0 0 0 0 // foo = 16

...

## Variable length indirect representation bit fields

For a variable length element utilizing a Huffman or variable length code table, an identical specification to the fixed length case shall be used.

The only difference is that the indices of the map are now of variable length and shall unambiguously identify each output value in addition of being unique (e.g. leaf values of a binary tree).

EXAMPLE 1 ⎯

class val {

unsigned int foo;

int bar;

}

map sample\_vlc\_map (val) {

0b0000.001, {0, 5},

0b0000.0001, {1, -14}

}

Due to the large number of possible entries in variable length code tables, it may be inefficient to keep using variable length codewords for all possible values. Because of this, they are often partially defined. This necessitates the use of escape codes, that signal the subsequent use of a fixed-length (or even variable length) value representation. To support this, parsable variable ***type*** declarations are allowed for map output values.

In this case, the type of an output *value* within the map declaration shall match the type associated with the map’s *output\_type*.

**Rule E.5: Maps with escape codes**

map map\_identifier (output\_type) {

index**,** {value | type(length)[**,** …] }[,

…]

}

In the following example, when the codeword 0b0000.0000.1 is encountered in the bitstream, then the value ‘5’ is assigned to the first element of the output *value* (i.e., val.foo). The following 32 bits are then parsed and assigned as the value of the second element of the output *value* (i.e., val.bar).

EXAMPLE 2 ⎯

class val {

unsigned int foo;

int bar;

}

map sample\_map\_with\_esc (val) {

0b0000.001, {0, 5},

0b0000.0001, {1, -14},

0b0000.0000.1, {5, int(6)},

0b0000.0000.0, {0, -20}

}

val(sample\_map\_with\_esc) myVal;

An example bitstream for this would be:

EXAMPLE 3 ⎯

...

0 0 0 0 0 0 0 0 1 // input 0b0000.0000.1 maps to

// myVal = { foo = 5,

0 1 0 0 0 0 // bar = 16 }

...

NOTE In case more than one element utilizes a parsable type declaration, the order in which elements are parsed determines the extracted values.

## Variable length strings

A variable length string shall be represented as:

Rule E.6: String data types

[const] [aligned[(modifier)]] string\_type string\_identifier [**=** [**u**|**u8**]**"**string\_value**"**]**;**

The string\_type may be any of the following: utf8string, utfstring, utf8list, base64string. The format of those string types are described in Table 1. In these descriptions, null-terminated means that the last character of a string is Unicode NUL, and hence an empty string is represented by a single Unicode NUL. Some fields using these types may restrict the characters permitted. In addition, space-separated means that a SPACE character whose Unicode is U+0020 is used as string separator. In a **utf8list** string, there shall not be any leading or trailing space character nor two consecutive space characters.

The keyword const has the same meaning as in subclause 6.2.5.

The keyword aligned and its modifier have the same meaning as in subclause 6.2.1.

1. String data type formats

|  |  |
| --- | --- |
| string\_type | **Encoded bitstream description** |
| utf8string | UTF-8 string as defined in RFC 3629, null-terminated. |
| utfstring | null-terminated string encoded using either UTF-8 or UTF-16.  If UTF-16 is used, the sequence of bytes shall start with a byte order mark (BOM) and the null termination shall be 2 bytes set to 0. |
| utf8list | null-terminated list of space-separated UTF-8 strings |
| base64string | null-terminated compliant base64 encoded data as defined in clause 4 of RFC 4648 |

EXAMPLE ⎯

aligned(8) utf8string message;

base64string encoded\_data;

## String value

The *string*\_*value* attribute shall represent a string literal. The *string\_value* shall not include a null termination character nor a BOM which may be present in the encoded bitstream. When a string literal is present, the encoding prefix of the string literal shall be compatible with the ***string\_type*** of the variable:

* a string literal prefix u8 with utf8string, utfstring and utf8list variables.
* a string literal prefix u with utfstring.

There shall not be any encoding prefix with base64string variables.

EXAMPLE 1 ⎯

utfstring code = u8"this is a code";

utfstring label = u"this is a UTF-16 label";

utf8list interesting\_list = u8"apple orange cherry";

utf8string mot = u8"cœur";

Parsable string variables with definitions that fall outside the flow of parsing shall have a default value of an empty string (e.g. "").

It is illegal to define a *string*\_*value* which conflicts with the string literal prefix or the ***string\_type***. For example, the following is an invalid specification:

EXAMPLE 2 ⎯

base64string = "inval?id"; // illegal character in a base64 string

# Composite data types

## Classes

Classes are the mechanism with which declarations of composite types is performed. Their syntax is as follows.

Rule C.1: Classes

[aligned[(modifier)]] class class\_identifier {

[element**;** …]

}

The keyword aligned and its modifier have the same meaning as in subclause 6.2.1.

The different *element* entries within the braces (“{“ and “}”) are the definitions of the contained elementary data types (as defined in clause 6), composite data types (as defined in clause 7) or syntactic flow control elements (as defined in clause 9). Furthermore, a particular member element declared in a class may be accessed (considering the scoping rules defined in 5.18) using the class member access operator (“**.**”).

Classes may also be encapsulated within other classes. In this case, the element in Rule C.1 is a class itself.

The order of declaration of the *elements* is the same order in which the elements appear in the bitstream.

NOTE As alignment is performed by advancing the current position in the bitstream to be before the next encoded value, it is somewhat redundant but not illegal to declare a class as aligned, and to declare the first element of the class as aligned. In this case the alignment before the first element’s encoded value in the bitstream would be performed using the largest of the two alignment declarations.

A class declaration shall not recursively reference itself.

The next rule describes the use of a declared class.

Rule C.2: Class data types

class\_identifier *class\_variable\_*identifier**;**

## Base and derived classes

The optional keyword extends followed by the base\_class attribute specifies that the class is a derived class and that it derives from another class of type base\_class called the base class. The base class is either a class (as defined in subclause 7.1), a derived class (as defined in this subclause) or an abstract class (as defined in subclause 7.3). Derivation implies that all information present in the base class can also be accessed in the derived class, and that, in the bitstream, all such information *precedes* any additional value definitions specified in the derived class.

RULE C.3: Derived classes

[aligned[(modifier)]] class class\_identifier [extends base\_class] {

[element**;** …]

}

The keyword aligned and its modifier have the same meaning as in subclause 6.2.1.

A derived class shall not differ in declared alignment from the base class.

In the following example, myBar.b immediately precedes myBar.c in the bitstream:

EXAMPLE 1 ⎯

class foo {

int(3) a;

}

class bar extends foo {

int(5) b;

int(10) c;

}

bar myBar;

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

// myBar = {

0 1 1 // a = 3,

0 0 1 0 0 // b = 4,

0 0 0 0 0 0 1 0 0 0 // c = 8,

// }

...

## Abstract classes

The optional keyword abstract specifies that the class is an abstract class. Only non-abstract classes, possibly derived from a base class, shall be present in the bitstream.

Rule C.4: Abstract classes

[aligned[(modifier)]] abstract class class\_identifier [extends base\_class] {

[element**;** …]

}

The keyword aligned and its modifier have the same meaning as in subclause 6.2.1.

The keyword **extends** and its *base\_class* have the same meaning as in subclause 7.2.

An abstract class may derive from another abstract class. An abstract class may also derive from a non-abstract class.

EXAMPLE 1 ⎯

abstract class Shape {

}

class Circle extends Shape {

unsigned int(16) radius;

}

class Rectangle extends Shape {

unsigned int(16) width;

unsigned int(16) height;

}

// only Circle and Rectangle classes can be present

class Example {

Circle c;

Rectangle r;

}

Example myExample;

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

// myExample = {

// c = {

0 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0 // radius = 2

// },

// r = {

0 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 // width = 4,

0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 // height = 8

// }

// }

...

## Polymorphism in class declaration

### General

If the bit keyword is added to a class definition, a derived class may appear at any point in the bitstream where its base class is used in the syntax, hence allowing to express polymorphism in an SDL specification.

Rule C.5: Class polymorphism

[aligned[(modifier)]] [abstract] class class\_identifier

[extends base\_class] [: bit(length) [class\_id\_identifier **=**] class\_id | id\_range |

*extended\_id\_range* ] {

[element**;** …]

}

The keyword aligned and its modifier have the same meaning as in subclause 6.2.1.

The keyword **abstract** has the same meaning as in subclause 7.3.

The keyword **extends** and its *base\_class* have the same meaning as in subclause 7.2.

The class\_id value is the key demultiplexing entity which is present in the bitstream before any class member variable of the class. This allows differentiation between base and derived classes when parsing. The length of the class\_id in bits is given by the length attribute following the bit keyword. The optional attribute *class*\_id\_identifier allows to access the class\_id from within the class. If the class is **aligned**, the alignment occurs in the bitstream before the encoded *class\_id*.

When the bit keyword is declared, all derived class class\_id attributes shall declare the same length attribute value as the base class.

When a *class\_id\_identifier* is declared, all derived classes shall declare the same *class\_id\_identifier*. If a *class\_id\_identifier* is not declared, all derived classes will also not declare one.

The actual class to be parsed is determined as follows:

* The base class declaration shall assign a constant value to class\_id.
* Each derived class declaration shall assign a constant value to class\_id.

Derived classes may use the same class\_id value as the base class. In that case, classes can only be discriminated through contextual information such as the value of a member variable from the base class.

EXAMPLE 1 ⎯

class Foo : bit(2) id = 0 {

// note that as "id" is declared it is accessible within this class

// as a constant value and lengthof(id) will return 2

int(5) a; // this a is preceded by the 2 bits of id

}

class Foo1 extends Foo : bit(2) id = 1 {

int(3) b; // this b is preceded by the 5 bits of a

}

class Foo2 extends Foo : bit(2) id = 2 {

int(4) c; // this c is preceded by the 5 bits of a

}

class Example {

Foo f; // may be Foo, Foo1 or Foo2

}

Example myExample;

// myExample.f.id will equal 0, 1 or 2

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

// myExample = {

// f = {

1 0 // id = 2,

0 0 0 0 1 // a = 1,

0 0 1 1 // c = 3

// }

// }

...

As an alternative to a single class\_id value, it is also possible to declare:

* an id\_range attribute which is a range of numerical values specified as start\_id **..** end\_id, inclusive of both bounds
* an *extended\_id\_range* attribute which is a combination of id\_range and class\_id specified as a comma-separated list of class\_id values and id\_range values e.g. myId=0x01,0x02,0x10..Ox1F.

In such cases:

* The base class declaration shall declare a range of valid values via an id\_range or *extended\_id\_range.*
* Each derived class declaration shall assign a constant value or a range of valid values via a class\_id value or an id\_range or *extended\_id\_range*. This value or range of values shall correspond to legal values declared for the base class.

EXAMPLE 3 ⎯

class Foo : bit(5) id = 1,10..20 {

int(5) a;

}

class Foo1 extends Foo : bit(2) id = 10 {

int(5) b; // this b is preceded by the 5 bits of a

}

It is possible that an SDL specification does not provide class declarations for all valid *class\_id* values. For example:

EXAMPLE 4 ⎯

class Foo : bit(3) 0,5,6..7 {

}

class Foo0 : bit(3) 0 {

}

class Foo5 : bit(3) 5 {

}

In the example above as there are no class declarations for the *class\_id* values of 6 and 7 this may result in undefined behavior. The behavior shall be specified by the standard presenting the SDL specification.

NOTE For example, the standard presenting the SDL specification could state that a class declared using an id\_range or *extended\_id\_range* is effectively abstract i.e. *class\_id* values encountered in the bitstream should only ever be values appearing as single *class\_id* values in derived class declarations. Alternatively, the standard could state that any *class\_id* encountered in the bitstream which is not associated with a derived class declaration but which is valid according to the base class declaration should result in the base class being used for parsing.

In the example above, if a bitstream is encoded using a *class\_id* value of 4 this will also result in undefined behavior and shall be addressed by the standard presenting the SDL specification.

### Polymorphism for abstract classes

Since abstract classes are not present in a bitstream, this means that the derived classes of an abstract class may use the entire range of *class\_id* values available. For the abstract base class, the class\_id shall be specified as 0 or alternatively an id\_range or *extended\_id\_range* shall be specified.

EXAMPLE ⎯

abstract class Foo : bit(1) id=0 { // the value 0 is not really used

}

// derived classes are free to use the entire range of ids, in this case 0..1

class Foo0 extends Foo : bit(1) id=0 {

int(5) a;

}

class Foo1 extends Foo : bit(1) id=1 {

int(10) b;

}

class Example {

Foo f; // can only be Foo0 or Foo1, not Foo

}

## Expandable classes

When the expandable keyword is used in the class declaration, it indicates that the class may contain implicit arrays or undefined trailing data, called the “expansion”.

Rule C.6: Expandable classes

[aligned[(modifier)]] expandable[(max\_class\_size)] [abstract] class class\_identifier

[extends base\_class] [**:** bit(length) [class\_id\_identifier **=**] class\_id |

id\_range | *extended\_id\_range*]{

[element**;** …]

}

The keyword aligned and its modifier have the same meaning as in subclause 6.2.1.

The keyword **abstract** has the same meaning as in subclause 7.3.

The keyword **extends** and its *base\_class* have the same meaning as in subclause 7.2.

The keyword **bit** and its *length* have the same meaning as in subclause 7.4.1.

Expandable classes may be used for classes that are required to support future compatible extensions or that may include private data. For example, a legacy device can decode an expandable class up to the last parsable variable appearing in the version of the class declaration that the device is aware of and can skip the unknown class data following the last known variable.

To this end, an expandable class explicitly encodes its own size in bytes in the bitstream. The size precedes any parsable variables of the class and its variable-length encoding is defined below using the SDL definition:

int sizeOfInstance = 0;

bit(1) nextByte;

bit(7) sizeByte;

sizeOfInstance = sizeByte;

while(nextByte) {

bit(1) nextByte;

bit(7) sizeByte;

sizeOfInstance = sizeOfInstance << 7 | sizeByte;

}

NOTE By definition, the encoding of sizeOfInstance is always an integer number of bytes in size.

The size information is implicitly accessible within the class as the member variable sizeOfInstance whenever a class is made expandable.

An expandable class may derive from another expandable class. An expandable class may also derive from a non-expandable class. A non-expandable class may not derive from an expandable class.

As the sizeOfInstance value encodes the class size in bytes, an expandable class should be declared in a way to ensure that its size is always an integer number of bytes. If this is not the case, padding bits will exist in the bitstream at the end of the last encoded parsable member variable. These padding bits will be included when determining the sizeOfInstance value.

If the class declaration uses the **bit** keyword (defined in subclause7.4.1) indicating the presence of an encoded class\_id, the encoding of this value shall precede the size encoding. If the class is aligned, the alignment occurs in the bitstream before the encoded *class\_id* and size information. The size information shall not include the number of bytes needed for the size encoding, the bits skipped to achieve alignment nor the *class\_id* encoding.

Anywhere in the bitstream where a set of expandable classes with class\_id values is expected, it is permissible to intersperse expandable classes with unknown class\_id values. These classes shall be skipped, using the size information.

The following example declares an expandable class with a maximum size of 120 bytes:

EXAMPLE 1 ⎯

aligned expandable(120) class Example {

// note that as the class is expandable "sizeOfInstance" is accessible within this

// class as a constant value and lengthof(sizeOfInstance) will return a multiple of 8

int(3) a;

// 5 padding bits will follow in the bitstream to ensure the class

// size is an integer number of bytes

}

Example myExample;

// myExample.sizeOfInstance == 1

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

<skip 0..7 bits> // aligned

0 // nextByte = 0

0 0 0 0 0 0 0 1 // sizeByte = 1

// myExample = {

// sizeOfInstance = 1,

0 0 1 // a = 1

0 0 0 0 0 // 5 padding bits to ensure myExample.sizeOfInstance == 1 byte

// }

...

## Parameter types

A parameter type declares a class with parameters. This addresses cases where the data structure of the class depends on referencing values of one or more other parsed items. Parameter types provide placeholders for such references, in the same way as the arguments in a C function declaration. The syntax of a class declaration including parameter functionality is as follows.

Rule C.7: Class parameter types

[aligned[(modifier)]] [expandable[(max\_class\_size)]] [abstract] class class\_identifier

[(parameter\_list)] [extends base\_class[(parameter\_value[, parameter\_value]…)]]

[**:** bit(length)[class\_id\_identifier**=**] class\_id | id\_range | *extended\_id\_range*] {

[element**;** …]

}

The keyword aligned and its modifier have the same meaning as in subclause 6.2.1.

The keyword **expandable** and its *max\_class\_size* have the same meaning as in subclause 7.5.

The keyword **abstract** has the same meaning as in subclause 7.3.

The keyword **extends** and its *base\_class* have the same meaning as in subclause 7.2.

The keyword **bit** and its *length* have the same meaning as in subclause 7.4.1.

The *parameter\_list* is a list of type or **class** identifiers and variable identifier pairs separated by commas. Usage of a ***string\_type*** in a *parameter\_list* is not supported.

A class that uses parameters is dependent on the values in its parameter list. When populating the member values of such a class, the parameters shall be already be populated with values. 8

The next rule describes the use of a declared class which uses parameters.

Rule C.8: Parameterized class data types

class\_identifier *class\_variable\_*identifier **(**parameter\_value [, parameter\_value]…**);**

In the following example class B declares two parameters a and i and the values are populated within class C when parsing the bitstream:

EXAMPLE 1 ⎯

class A {

unsigned int(4) format;

}

class B (A a, int i) { // B declares two parameters

unsigned int(i) foo;

if (a.format == 2) {

int(4) bar;

}

}

class C {

int(2) i;

A a;

B b(a, i); // parameters with populated values are provided

}

C c;

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

// c = {

1 1 // i = 3,

// a = {

0 0 1 0 // format = 2

// },

// b = {

1 0 1 // foo = 5,

0 1 1 1 // bar = 7

// }

...

In the following example class A declares one parameter i and the value is populated during the declaration of class B:

EXAMPLE 3 ⎯

class A (int i) { // A uses one parameter

unsigned int(4) id = i; // parsed value for id must equal the value of i

unsigned int(4) value1;

}

class B extends A(3) { // parameter value provided

unsigned int(4) value2;

}

B b;

An example bitstream for this would be:

EXAMPLE 4 ⎯

...

// b = {

1 1 // id = 3,

0 0 1 0 // value1 = 2,

0 1 0 1 // value2 = 5,

// }

...

In the following example class A declares two parameters i and j which are then further exposed as parameters for the derived class B. The value of i is populated during the declaration of class C and the value of j is populated during the definition of the class variable c:

EXAMPLE 5 ⎯

class A (int i, int j) { // A uses two parameters

unsigned int(i) id = j; // width of id is i and the value must equal j

}

class B (int i, int j) extends A(i, j) { // two parameter values propagated

unsigned int(4) value1;

}

class C (int j) extends B(4, j) { // one parameter value provided, one propagated

unsigned int(4) value2;

}

C c(3); // one parameter value provided

An example bitstream for this would be:

EXAMPLE 6 ⎯

...

// c = {

0 0 1 1 // id = 3, (width of id is 4, id must equal 3)

0 0 1 0 // value1 = 2,

0 1 0 1 // value2 = 5,

// }

...

In the following example class A declares one parameter i and class B declares one parameter j whilst class C declares two parameters x and y. The class C parameters are used to both define its base class A and its member class B:

EXAMPLE 7 ⎯

class A (int i) { // A uses one parameter

unsigned int(i) value1; // width of value1 is i

}

class B (int j) { // B uses one parameter

unsigned int(j) value2; // width of value2 is j

}

class C (int x, int y) extends A(x) { // one parameter value propagated, one used

B b(y);

}

C c(4, 2); // two parameter values provided

An example bitstream for this would be:

EXAMPLE 8 ⎯

...

// c = {

0 1 0 1 // value1 = 5,

// b = {

1 1 // value2 = 3

// }

// }

...

[Editor’s Note: Previously proposed but not accepted:

The SDL does not specify whether parameter values should be considered as passed by value or passed by reference. Care should be taken when creating SDL specifications to avoid unspecified behavior. As an example:

EXAMPLE ⎯

class A {

int(2) i;

}

class B (A a) {

int(2) j;

a.i++;

}

A a;

// assume a.i == 2

B b(a);

// does a.i == 2 or 3 here?

An example bitstream for this would be:

EXAMPLE 8 ⎯

...

// a = {

1 0 // i = 2,

// },

// b = {

1 1 // j = 3

// }

// does a.i == 2 or 3 here?

...

A computer program implementation or a standard containing an SDL specification may choose to specify such details.]

Parameter values used to populate a class variable shall match the number and type of the parameters specified in the parameter list. For example the following is an invalid specification:

EXAMPLE 9 ⎯

class A (int i, int j) {

unsigned int(i) foo;

unsigned int(j) bar;

}

class B {

int(2) i;

float(64) j;

A a1(i, j); // illegal mismatch of parameter types

A a2(i); // illegal mismatch of parameter count

}

The evaluation order of parameter values is not defined by the SDL which may lead to undefined behavior. A computer program implementation or a standard presenting an SDL specification may choose to define expected behavior in these scenarios. For example:

EXAMPLE 10 ⎯

class A (int i, int j) {

unsigned int(i) foo;

unsigned int(j) bar;

}

class B {

int(2) i;

A a1(i++, i++); // this may result in A(1, 2) or A(2, 1)

}

## Arrays

### General

Arrays are defined using square brackets. The array definition is applicable to both elementary types and classes.

Rule A.1: Arrays

[aligned[(modifier)]] *typespec* array\_identifier[length]**;**

*typespec* is a parsable type specification (e.g., an elementary typewith a *length* attribute, e.g. ‘int(2)’) or a **class** identifier. The attribute length specifies the capacity of the array. The length value can depend on other bitstream values or expressions that involve such values.

The *length* value shall be an integer value of 0 or greater. The maximum *length* value is explicitly identified as an unspecified constraint.

In the following example a is an array of 5 elements, each of which is represented using 4 bits in the bitstream and interpreted as an unsigned integer:

EXAMPLE 1 ⎯

unsigned int(4) a[5];

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

// a = [

0 0 0 1 // 1,

0 0 1 0 // 2,

0 0 1 1 // 3,

0 1 0 0 // 4,

0 1 0 1 // 5

// ]

...

In the following example the length of c depends on the actual value of b:

EXAMPLE 3 ⎯

int(10) b;

int(2) c[b];

An example bitstream for this would be:

EXAMPLE 4 ⎯

...

0 0 0 0 0 0 0 0 1 0 // b = 2

// c = [

0 1 // 1,

2 0 // 2

// ]

...

Individual values of an array are accessed using the array element access operator (e.g. a matching pair of square brackets ‘**[**‘ and ‘**]**’). For example:

EXAMPLE 5 ⎯

int(4) a[5];

int b = a[0]; // access the value of the first element in array a

a[1] = 10; // set the value of the second element in array a

NOTE In the example above b is a non-parsable variable as defined in 8.

### Alignment

The keyword aligned indicates that the start of the array data is aligned on a byte boundary.

For example, an array where the first item is aligned on a byte boundary and the individual items are packed with no skipped bits for alignment:

EXAMPLE 1 ⎯

aligned bit(5) foo[7];

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

<skip 0..7 bits> // aligned

// foo = [

0 0 0 0 1 // 1,

0 0 0 1 0 // 2,

0 0 0 1 1 // 3,

0 0 1 0 0 // 4,

0 0 1 0 1 // 5,

0 0 1 1 0 // 6,

0 0 1 1 1 // 7

// ]

...

An optional numeric modifier attribute may be used to signify alignment on boundaries other than byte. Allowed values are 8, 16, 32, 64, and 128. Any skipped bits due to alignment shall have the value ‘0’.

## Multi-dimensional arrays

Multi-dimensional arrays are also supported. The parsing order from the bitstream corresponds to scanning the array by incrementing first the right-most index of the array, then the second, and so on.

Rule A.2: Multi-dimensional arrays

[aligned[(modifier)]] *typespec* array\_identifier [length][length]…**;**

The keyword aligned and its modifier have the same meaning as in subclause **Error! Reference source not found.**.

The construct *typespec* has the same meaning as in subclause 7.7.

In the following example, a is an array of 2 elements, each of which is represented as an array of 3 elements using 4 bits in the bitstream and interpreted as an unsigned integer:

EXAMPLE 1 ⎯

unsigned int(4) a[2][3];

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

// a = [

0 0 0 1 0 0 1 0 0 0 1 1 // [ 1, 2, 3 ],

0 1 0 0 0 1 0 1 1 1 0 0 // [ 4, 5, 6 ]

// ]

...

The maximum number of dimensions for an array is explicitly identified as an unspecified constraint.

Individual values of a multi-dimensional array are accessed using a sequence of array element access operators (e.g. a matching pair of square brackets ‘**[**‘ and ‘**]**’) where the number of operators matches the dimensionality of the array. For example:

EXAMPLE 3 ⎯

int(4) a[5][7];

int b = a[1][2]; // access the value in the second row and third column of array a

a[2][3] = 10; // set the value in the third row and fourth column of array a

## Partial arrays

In several situations, it is desirable to load the values of an array one by one, to check for a terminating or other condition. For this purpose, an extended array definition is allowed in which individual elements of the array may be accessed for population i.e. allowing the definition of dynamically sized sparse arrays.

Rule A.3: Partial arrays

[aligned[(modifier)]] *typespec* array\_identifier[[index]]**;**

The keyword aligned and its modifier have the same meaning as in subclause **Error! Reference source not found.**.

The construct *typespec* has the same meaning as in subclause 7.7.

Here index is the index of the element of the array that is defined. Index values start at the value 0. Several such partial definitions may be given, but they shall all agree on the typespec specification.

s

Multidimensional partial arrays are also supported.

Rule A.4: Partial multi-dimensional arrays

[aligned[(modifier)]] *typespec* array\_identifier [[length]][[length]]…**;**

The keyword aligned and its modifier have the same meaning as in subclause **Error! Reference source not found.**.

The construct *typespec* has the same meaning as in subclause 7.7.

The following example indicates the element a(5, 3) of the array i.e. the element in the 6th row (row index 5) and the 4th column (column index 3):

EXAMPLE 1 ⎯

int(4) a[[5]][[3]];

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

// a = [

// [ , , , ],

// [ , , , ],

// [ , , , ],

// [ , , , ],

// [ , , , ],

0 1 0 1 // [ , , , 5 ]

// ]

...

The following example indicates the entire 6th row (row index 5) of the array which contains 3 elements each of which is a 4-bit integer value:

EXAMPLE 3 ⎯

int(4) a[[5]][3];

An example bitstream for this would be:

EXAMPLE 4 ⎯

...

// a = [

// [ , , ],

// [ , , ],

// [ , , ],

// [ , , ],

// [ , , ],

0 0 0 1 0 0 1 1 0 1 0 1 // [ 1, 3, 5 ]

// ]

...

The following example indicates the entire 4th column (column index 3) of the array which contains 5 rows:

EXAMPLE 5 ⎯

int(4) a[5][[3]];

An example bitstream for this would be:

EXAMPLE 6 ⎯

...

// a = [

0 0 0 1 // [ , , , 1 ],

0 0 1 0 // [ , , , 2 ],

0 0 1 1 // [ , , , 3 ],

0 1 0 0 // [ , , , 4 ],

0 1 0 1 // [ , , , 5 ]

// ]

...

In the following example, the entries of the one-dimensional array wordLength are populated in a sparse manner within the **for**-loop (**for**-loops are presented in 9.2). On the subsequent line, these values are then accessed to define and populate the two-dimensional array words (again in a sparse manner for one of the dimensions):

EXAMPLE 7 ⎯

unsigned int(8) wordCount;

int i;

for (i = 0; i < wordCount; i++) {

// sparse array population

unsigned int(8) wordLength[[i]];

// first dimension sparse array population,

// second dimension fully populated with a size defined by wordLength[i]

bit(8) words[[i]][wordLength[i]];

}

An example bitstream for this would be:

EXAMPLE 8 ⎯

...

0 0 0 0 0 0 1 1 // wordCount = 3

0 0 0 0 0 0 1 0 // wordLength[0] = 2

// words[0] = [

0 0 0 0 0 1 1 1 // 7,

0 0 0 0 1 0 0 0 // 8

// ]

0 0 0 0 0 0 1 1 // wordLength[1] = 3

// words[1] = [

0 0 0 0 0 0 0 1 // 1,

0 0 0 0 0 0 0 1 // 1,

0 0 0 0 0 0 0 1 // 1,

// ]

0 0 0 0 0 0 1 0 // wordLength[2] = 1

// words[2] = [

0 0 0 0 0 1 1 0 // 6

// ]

...

## Implicit arrays

An array with an implicit length is an implicit array. It is indicated by an array definition without a length specification. To limit the possible minimum and maximum implicit length of the array, an optional *range* specification may be used as the specification of the length.

Rule A.5: Implicit arrays

[aligned[(modifier)]] *typespec* array\_identifier [[range]];

The keyword aligned and its modifier have the same meaning as in subclause 6.2.1.

The construct *typespec* has the same meaning as in subclause 7.7.

Multidimensional implicit arrays are not supported.

When a series of polymorphic classes is present in the bitstream, it may be represented as an array of the same type as that of the base class.

For example, assume that a set of polymorphic classes is declared, derived from the base class Foo (which may or may not be abstract):

EXAMPLE 1 ⎯

class Foo : bit(16) id = 0 {

int(5) a;

}

Then an array of polymorphic classes with an explicit length of 100 can be defined as follows:

EXAMPLE 2 ⎯

Foo explicit\_length\_array[100];

For an array of such classes, it is possible to implicitly determine the length by examining the validity of the class\_id of the class. Classes are inserted in the array as long as the class\_id can be properly resolved to one of the class\_id values declared in the base class (if not abstract) or its derived classes.

NOTE When an implicit array is followed by further parsable data, there is a possibility that the following encoded bits match the value of a class class\_id. In this case, the SDL specification author should take care of preventing this possible ambiguity in the bitstream by a mechanism such as emulation prevention bytes or reserved value markers.

In the following example, the number of elements is implicitly obtained via class\_id resolution:

EXAMPLE 3 ⎯

Foo f[]; // length implicitly obtained via class\_idresolution

In the following example, f may have at least 1 and at most 255 elements:

EXAMPLE 4 ⎯

Foo f[1..255]; // at least 1, at most 255 elements

# Non-parsable variables

## General

To accommodate complex syntactic constructs, in which context information cannot be directly obtained from the bitstream but only as a result of a computation, non-parsable variables are allowed.

Non-parsable variables may be used in expressions and conditions in the same way as parsable variables. In the following example, the number of non-zero elements of an array is computed.

EXAMPLE 1 ⎯

int(4) myArray[100]; // parsable array of 100 integers, each 4 bits in size

int i; // non-parsable integer variable

int n = 0; // non-parsable integer variable

for (i = 0; i < 100; i++) {

if (myArray[i] != 0) {

n++;

}

}

// parse as many coefficients as there are non-zero elements in myArray

int(16) coefficients[n];

## Elementary data types

Non-parsable elementary data type variable definitions are distinguished from parsable variable definitions due to the absence of a *length* attribute.

Rule NP.1: Elementary data types

[const] type identifier [**=** value]**;**

The keyword const has the same meaning as in subclause 6.2.5 and **type** is as previously defined in 6.2.2.

For example, an unsigned integer non-parsable variable would be represented as:

EXAMPLE ⎯

unsigned int foo;

The result of accessing the value of a non-parable variable which has not yet been initialised with a value via the assignment operator (e.g. ‘**=**’) is not defined by the SDL any may lead to undefined behavior.

### Value

The value attribute shall be present only when the variable should be initialised with a value. A variable may be used when defining a value.

For example:

EXAMPLE ⎯

unsigned int initial = 10;

unsigned int foo = initial;

NOTE Unlike a parsable variable, a range of values cannot be specified when declaring a non-parsable variable.

Refer to 5.12.3 for a discussion on the limits of *value*.

Refer to 5.12.4 for a discussion on the coercion of *value*.

## Arrays

Arrays are defined using square brackets. The array definition for non-parsable variables is only applicable to elementary types.

Rule NP.2: Arrays

*type* array\_identifier [length]**;**

The attribute length has the same meaning as in subclause 7.7 and the construct ***type*** is as previously defined in 6.2.2.

In the following example a is an array of 5 elements, each of which is an unsigned integer:

EXAMPLE 1 ⎯

unsigned int a[5];

In the following example the length of c depends on the value of the parsable variable b:

EXAMPLE 2 ⎯

int(10) b;

int c[b];

Individual values of an array are accessed using the array element access operator (e.g. a matching pair of square brackets ‘**[**‘ and ‘**]**’).. For example:

EXAMPLE 3 ⎯

int a[5];

int b = a[0]; // access the value of the first element in array a

a[1] = 10; // set the value of the second element in array a

NOTE The SDL does not provide a syntax for initialisation of all elements of an array using a literal value.

## Multi-dimensional arrays

Non-parsable multi-dimensional arrays are also supported.

Rule NP.3: Multi-dimensional arrays

*typespec* array\_identifier [length][length]…**;**

The attribute length has the same meaning as in subclause 7.7 and the construct ***type*** is as previously defined in 6.2.2.

In the following example, a is an array of 5 rows, each of which is represented as an array of 6 unsigned integer values:

EXAMPLE ⎯

unsigned int a[5][6];

The maximum number of dimensions for an array is explicitly identified as an unspecified constraint.

Individual values of a multi-dimensional array are accessed using a sequence of array element access operators (e.g. a matching pair of square brackets ‘**[**‘ and ‘**]**’) where the number of operators matches the dimensionality of the array. For example:

EXAMPLE 3 ⎯

unsigned int a[1][2];

int b = a[1][2]; // access the value in the second row and third column of array a

a[2][3] = 10; // set the value in the third row and fourth column of array a

# Syntactic flow control

Syntactic flow control provides constructs that allow conditional parsing, depending on context, as well as repetitive parsing.

## Conditionals

The **if-**then**-else** construct is used for testing conditions.

Rule FC.1: Flow control using if-then-else

if (condition)

…

[ else if (condition)

…]

[else

…]

Each *condition* is evaluated sequentially in order until a *condition* evaluates to a logic value of true. When this occurs, the conditional clause associated with it is evaluated and then the remainder of the construct is skipped. The ability to specify a final **else** statement without a condition allows for a default case where none of the explicit conditions are satisfied.

Each conditional clause must have at least one statement. If a conditional clause contains a single statement, then usage of braces ‘**{**’ and ‘**}**’ is optional (they are implicit), but if multiple statements are present then braces are required. For example:

EXAMPLE 1 ⎯

if (condition)

int(8) a;

else {

int(8) b;

int(8) c;

}

Because an **else** statement is optional there is potential for ambiguity when an **else** is omitted within a nested **if**-then-**else** construct. In this case, the else statement shall be associated with the closest previous if statement that did not have an if. For example:

EXAMPLE 2 ⎯

if (condition1)

if (condition2)

int(8) a;

else // associated with "if (condition2)"

int(8) b;

In the following example, the presence of the parsable variable bar is determined by the bar\_flag:

EXAMPLE 3 ⎯

class conditional\_class1 {

unsigned int(3) foo;

bit(1) bar\_flag;

if (bar\_flag) {

unsigned int(8) bar;

}

unsigned int(4) more\_foo;

}

conditional\_class1 myExample1;

An example bitstream for this would be:

EXAMPLE 4 ⎯

...

// myExample1 = {

0 0 1 // foo = 1,

1 // bar\_flag = 1,

0 0 0 1 0 0 0 0 // bar = 16,

0 1 0 0 // more\_foo = 4

// }

...

NOTE The use of bar\_flag necessitates its definition before the conditional is encountered.

A parsable variable may be defined more than once across conditional branches if the defined ***type*** is identical (the *length* attribute may differ). In the following example, two different representations for the parsable variable bar are defined, depending on the value of bar\_flag.

EXAMPLE 5 ⎯

class conditional\_class2 {

unsigned int(3) foo;

bit(1) bar\_flag;

if (bar\_flag) {

unsigned int(8) bar;

}

else {

unsigned int(4) bar;

int(8) optional\_foo;

}

unsigned int(4) more\_foo;

}

conditional\_class2 myExample2;

An example bitstream for this would be:

EXAMPLE 6 ⎯

...

// myExample2 = {

0 0 1 // foo = 1,

0 // bar\_flag = 0,

0 1 0 0 // bar = 4,

0 0 0 0 0 1 1 1 // optional\_foo = 7

0 1 0 0 // more\_foo = 4

// }

...

As presented in 5.18.1, parsable variables defined within a class declaration, regardless of conditional branch scopes have class scope, i.e., they are available as class member variables. Using the above class declaration, the following example clarifies this further:

EXAMPLE 7 ⎯

conditional\_class2 c;

// c.foo, c.bar\_flag, c.bar, c.optional\_foo and c.more\_foo are always all accessible

// if c.bar\_flag == 1 then:

// c.bar is an 8-bit unsigned integer with a value parsed from the bitstream

// c.optional\_foo is an 8-bit integer with no parsed value and a default value of 0

// if c.bar\_flag == 0 then:

// c.bar is a 4-bit unsigned integer with a value parsed from the bitstream

// c.optional\_foo is an 8-bit integer with a value parsed from the bitstream

NOTE Usage of the keyword **break** to exit a clause within an **if**-then-**else** construct is not supported.

It is illegal to define **if**-then-**else** clauses which will result in duplicate class member variables being declared simultaneously. For example, the following is an invalid specification as it allows for the member variable foo to be declared more than once simultaneously:

EXAMPLE 8 ⎯

class invalid\_class {

int(2) bar;

if (bar > 0) {

unsigned int(8) foo; // first declaration of foo if bar > 0

}

if (bar > 1) {

unsigned int(4) foo; // second declaration of foo which will

// conflict with the first declaration if bar > 1

}

}

To facilitate cascades of **if**-then-**else** constructs, the **switch** syntax is also allowed.

Rule FC.2: Flow control using switch

switch (condition) {

[case label:  
 [… ]

[break;]]

…

[default:

[…]]

}

The condition in a **switch** statement shall evaluate to an integer value. Each *label* should be an integer constant expression and each *label* in the same **switch** statement shall be unique.

EXAMPLE 9 ⎯

unsigned int(32) code;

switch (code) {

case 0:

Foo f;

break;

case 1:

Bar b;

break;

default:

Moo m;

}

If the **break** keyword is not encountered, then the flow will continue onto the next labelled clause. The **case** and the **default** clauses may be optionally placed in braces ‘**{**’ and ‘**}**’ and this has no effect on the conditional flow. For example:

EXAMPLE 10 ⎯

unsigned int(32) code;

switch (code) {

case 0:

// flow through to case 1

case 1: {

Foo f1;

} // brace scope has no effect

Foo f2;

// flow through to case 2

case 2: {

Bar b1;

Bar b2;

break;

} // brace scope has no effect

// no flow through to default

default:

Moo m;

}

As with **if-**then**-else** construct, parsable variables may be defined more than once across conditional branches in a **switch** statement if the defined ***type*** is identical (the *length* attribute may differ). Additionally it is illegal to define **switch** clauses which will result in duplicate class member variables being declared simultaneously.

## Loops

Context-sensitive constructs are also provided for iterative parsing. These constructs imply the repetitive use of the same syntax to parse the bitstream, until some condition is met.

In the **for**-loop syntax, expression1, expression2 and expression3 each constitute a single statement. If specified, expression1 can be either a non-parsable variable definition with an assigned value or a value assignment and is evaluated prior to starting the repetitions. Then if expression2 is not specified, or it is specified and it evaluates to a logic value of true, the statements within the scope of the **for**-loop are processed, followed by the evaluated of expression3 if it is specified. The process repeats until expression2 evaluates to a logic value of false.

Rule FC.3: Flow control using for

for ([expression1]; [expression2]; [expression3])

…

The following example presents using a **for**-loop for repeated parsing of values from a bitstream:

EXAMPLE 1 ⎯

class Looped {

int(8) multiplier;

int(4) count;

int i;

int values[count];

for (i = 0; i < count; i++) {

int(8) offset;

values[i] = multiplier \* offset;

}

}

Looped looped;

// here, looped.offset will be the last offset value parsed from the bitstream

An example bitstream for this would be:

EXAMPLE 2 ⎯

...

// looped = {

0 0 0 0 0 1 0 0 // multiplier = 4,

0 0 1 0 // count = 2,

0 0 0 0 0 1 1 0 // offset = 6 (when i == 0)

0 0 0 0 0 0 1 0 // offset = 2 (when i == 1)

// }

// here looped.offset == 2 and looped.values == [ 24, 8 ]

...

The body of the **for**-loop must specify at least one statement. If a single statement is specified, then usage of braces ‘**{**’ and ‘**}**’ is optional (they are implicit), but if multiple statements are present then braces are required. For example:

EXAMPLE 3 ⎯

int i;

int n = 0;

for (i = 0; i < 10; i++) {

n++;

n--;

}

// n == 0 here

for (i = 0; i < 10; i++)

n++;

n--; // this is outside of the for loop

// n == 9 here

The following example shows a non-parsable variable defined within the scope of the **for**-loop construct:

EXAMPLE 4 ⎯

int n = 0;

for (int i = 0; i < 10; i++) {

n++;

// i is accessible here

}

// i is not accessible here

NOTE Usage of the keyword **break** to exit a for-loop construct is not supported.

In the **do-while**-loop syntax, the block of statements is evaluated until condition evaluates to a logical value of false.

NOTE The block will be evaluated at least once.

Rule FC.4: Flow control using do

do {

…

} while (condition);

The body of the **do-while**-loop must specify at least one statement. Braces ‘**{**’ and ‘**}**’ are always required.

EXAMPLE 5 ⎯

int i = 10;

do {

i--;

} while (i > 0);

NOTE Usage of the keyword **break** to exit a do-while-loop construct is not supported.

In the **while**-loop syntax, the loop is evaluated zero or more times, as long as condition evaluates to a logic value of true.

Rule FC.5: Flow control using while

while (condition)

…

The body of the **while**-loop must specify at least one statement. If a single statement is specified, then usage of braces ‘**{**’ and ‘**}**’ is optional (they are implicit), but if multiple statements are present then braces are required. For example:

EXAMPLE 6 ⎯

int i = 10;

while (i > 0) {

i++;

i = i - 2;

}

// i < 0 here

i = 10

while (i > 0)

i++;

i = i - 2; // this is outside of the while loop and is never reached

NOTE Usage of the keyword **break** to exit a while-loop construct is not supported.

1. (normative)  
     
   SDL syntax

[Editor’s note: add formal EBNF grammar.]

1. (informative)  
     
   SDL user guide
   1. Getting started

In this section, we are going to cover some basic concepts to describe a binary structure using SDL. We will take the example of the MPEG-2 transport packet and construct a possible SDL specification to describe it. The binary structure of a transport packet according to ISO/IEC 13818-1 is described in Table 1.

Table 1 – ISO/IEC 13818-1 transport packet binary structure

| Syntax | No. of bits | Mnemonic |
| --- | --- | --- |
| transport\_packet(){ |  |  |
| **sync\_byte** | **8** | **bslbf** |
| **transport\_error\_indicator** | **1** | **bslbf** |
| **payload\_unit\_start\_indicator** | **1** | **bslbf** |
| **transport\_priority** | **1** | **bslbf** |
| **PID** | **13** | **uimsbf** |
| **transport\_scrambling\_control** | **2** | **bslbf** |
| **adaptation\_field\_control** | **2** | **bslbf** |
| **continuity\_counter** | **4** | **uimsbf** |
| if(adaptation\_field\_control = = '10' || adaptation\_field\_control = = '11'){ |  |  |
| adaptation\_field() |  |  |
| } |  |  |
| if(adaptation\_field\_control = = '01' || adaptation\_field\_control = = '11') { |  |  |
| for (i = 0; i < N; i++){ |  |  |
| **data\_byte** | **8** | **bslbf** |
| } |  |  |
| } |  |  |
| } |  |  |

As shown in Table 1, a “transport\_packet” is declared as a logical structure which can be encoded in a bitstream. A convenient way to represent the same logical structure in SDL is to declare a **class**. The minimum declaration for a class is an identifier and an empty body:

class transport\_packet {

// to be defined

}

The body of the class corresponds to the representation of the sequence of bits to be described. Based on Table 1, we can see that the first byte of a transport packet is the “sync\_byte”. Let’s then define this first byte in our new class:

class transport\_packet {

unsigned int(8) sync\_byte;

}

We chose to define our first variable as unsigned integer and call it sync\_byte. Since its length in the bitstream is one byte, we pass the number 8 in parenthesis as the length attribute of this variable. As the ISO/IEC 13818-1 standard requires the “sync\_byte” to be equal to 0x47, we can also define that this variable will always be equal to this value:

class transport\_packet {

unsigned int(8) sync\_byte = 0x47;

}

The “sync\_byte” is followed by three 1-bit flag elements: “transport\_error\_indicator”, “payload\_unit\_start\_indicator”, and “transport\_priority”. SDL syntax allows such raw binary data elements to be represented using the keyword bit (alternatively, we could also use unsigned int(1)):

class transport\_packet {

unsigned int(8) sync\_byte = 0x47;  
 bit(1) transport\_error\_indicator;

bit(1) payload\_unit\_start\_indicator;

bit(1) transport\_priority;

}

Following the flags is the packet identifier (“PID”) which is a 13-bit field. Since we may want to manipulate this variable as a number (PID has well-defined hexadecimal values), we can define it as an unsigned integer:

class transport\_packet {

unsigned int(8) sync\_byte = 0x47;  
 bit(1) transport\_error\_indicator;

bit(1) payload\_unit\_start\_indicator;

bit(1) transport\_priority;

unsigned int(13) PID;

}

After the PID comes two 2-bit control elements whose values are defined as bit strings, e.g. “00 = Not scrambled”, “01 = User-defined”, etc. We will thus also define those variables as **bit** variables:

class transport\_packet {

unsigned int(8) sync\_byte = 0x47;  
 bit(1) transport\_error\_indicator;

bit(1) payload\_unit\_start\_indicator;

bit(1) transport\_priority;

unsigned int(13) PID;  
 bit(2) transport\_scrambling\_control;

bit(2) adaptation\_field\_control;

}

At this point, we have declared the non-variable part of the transport packet. As shown in Table 1, the reminder of the packet depends on the value of the “adaptation\_field\_control” element. To represent this, we can use an SDL **if**-statement:

class transport\_packet {

unsigned int(8) sync\_byte = 0x47;  
 bit(1) transport\_error\_indicator;

bit(1) payload\_unit\_start\_indicator;

bit(1) transport\_priority;

unsigned int(13) PID;  
 bit(2) transport\_scrambling\_control;

bit(2) adaptation\_field\_control;

if (adaptation\_field\_control == 0b10 || adaptation\_field\_control == 0b11){

// to be defined

}

if (adaptation\_field\_control == 0b01 || adaptation\_field\_control == 0b11) {  
 // to be defined

}

}

As the reader can see, SDL allows us to test the value of a variable using the relational equal **==** operator. Here since they are bit variables we use the binary value literal representation with prefix ‘**0b**’. Finally, the logical combination of tests is achieved using the logical OR operator ‘**||**’.

All that remains to specify is the two clauses of the **if**-statements. In the first one, a logical structure “adaptation\_field()” is present. To represent this, let’s assume that we have previously declared an SDL class adaptation\_field, so that we define a variable whose type is this class:

class transport\_packet {

unsigned int(8) sync\_byte = 0x47;  
 bit(1) transport\_error\_indicator;

bit(1) payload\_unit\_start\_indicator;

bit(1) transport\_priority;

unsigned int(13) PID;  
 bit(2) transport\_scrambling\_control;

bit(2) adaptation\_field\_control;

if (adaptation\_field\_control == 0b10 || adaptation\_field\_control == 0b11) {  
 adaptation\_field data;

}

if (adaptation\_field\_control == 0b01 || adaptation\_field\_control == 0b11) {  
 // to be defined

}

}

Note that we don’t need to specify how large this data field is as long as the declaration of the adaptation\_field class determines it.

To complete the transport packet, we need to specify the clause of the second if-statement. The ISO/IEC 13818-1 standard defines this part of the transport packet as a loop over N-1 elements corresponding to a sequence of N-1 bytes. The SDL author may be tempted to use a for loop:

class transport\_packet {

unsigned int(8) sync\_byte = 0x47;  
 bit(1) transport\_error\_indicator;

bit(1) payload\_unit\_start\_indicator;

bit(1) transport\_priority;

unsigned int(13) PID;  
 bit(2) transport\_scrambling\_control;

bit(2) adaptation\_field\_control;

if (adaptation\_field\_control == 0b10 || adaptation\_field\_control == 0b11) {  
 adaptation\_field data;

}

if (adaptation\_field\_control == 0b01 || adaptation\_field\_control == 0b11) {  
 for (int i = 0; i < N; i++) {

bit(8) data\_byte;

}

}

}

However, this declaration may not yield the intended result – declaring the same variable N-1 times will overwrite all previously parsed values and only keep the last one. If the SDL parser does not need those values this may be acceptable, however in this scenario the data constitutes the payload of the transport packet, and we intend to keep this data intact. An elegant way to specify such a sequence of bytes is by declaring an array of N-1 elements instead of using a **for**-loop:

class transport\_packet {

unsigned int(8) sync\_byte = 0x47;  
 bit(1) transport\_error\_indicator;

bit(1) payload\_unit\_start\_indicator;

bit(1) transport\_priority;

unsigned int(13) PID;  
 bit(2) transport\_scrambling\_control;

bit(2) adaptation\_field\_control;

if (adaptation\_field\_control == 0b10 || adaptation\_field\_control == 0b11) {  
 adaptation\_field data;

}

if (adaptation\_field\_control == 0b01 || adaptation\_field\_control == 0b11) {

bit(8) data\_byte[N-1];

}

}

Our SDL class declaration of a “transport\_packet” class is almost complete. There is still an unknown variable, N, in use. The ISO/IEC 13818-1 standard defines N as “specified by 184 minus the number of bytes in the adaptation\_field()”. The variable N is thus not written in the bitstream but computed based on the context of the class at this point of the binary structure. For this, SDL defines non-parsable variables which can hold any value for the purpose of declaring SDL elements. Let’s then define this variable as an unsigned integer and initialise it with the value 184:

class transport\_packet {

unsigned int(8) sync\_byte = 0x47;  
 bit(1) transport\_error\_indicator;

bit(1) payload\_unit\_start\_indicator;

bit(1) transport\_priority;

unsigned int(13) PID;  
 bit(2) transport\_scrambling\_control;

bit(2) adaptation\_field\_control;

unsigned int N = 184;

if (adaptation\_field\_control == 0b10 || adaptation\_field\_control == 0b11) {  
 adaptation\_field data;  
 N = N – 1 – data.adaptation\_field\_length;

}

if (adaptation\_field\_control == 0b01 || adaptation\_field\_control == 0b11) {

bit(8) data\_byte[N];

}

}

In the class adaption\_field, we assume the existence of the variable adaptation\_field\_length which is defined in the ISO/IEC 13818-1 standard as “an 8-bit field specifying the number of bytes in the adaptation\_field immediately following the adaptation\_field\_length”. Therefore, we can access this variable with the SDL class member access operator ‘**.**’ and calculate the size of the array.

The specification of the MPEG-2 transport packet as an SDL class declaration is now complete.

[Editor’s note: Addressing scope and entry point clarity, it was recently proposed to add:

A potential class declaration and definition providing a global entry point to parse an endless stream of transport packets might be:

class transport\_stream {

while (true) {

transport packet packet;

}

}

transport\_stream stream;]