**ISO/IEC 14496-34:22##(X)**

ISO TC 1/SC 29/WG 03

Date: 2023-09-20

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

Pot. Impr. DIS 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](#_Toc146119011)

[Introduction vi](#_Toc146119012)

[1 Scope 1](#_Toc146119013)

[2 Normative references 1](#_Toc146119014)

[3 Terms and definitions 1](#_Toc146119015)

[4 Rule conventions 1](#_Toc146119016)

[4.1 Rule formatting 1](#_Toc146119017)

[4.2 Rule tokens 2](#_Toc146119018)

[5 General syntax aspects 2](#_Toc146119019)

[5.1 Operators 2](#_Toc146119020)

[5.2 Whitespace 3](#_Toc146119021)

[5.3 Built-in operators 3](#_Toc146119022)

[5.4 Endianness 3](#_Toc146119023)

[5.5 Representation of numbers 3](#_Toc146119024)

[5.6 Comments 3](#_Toc146119025)

[5.7 Binary values 3](#_Toc146119026)

[5.8 Hexadecimal values 4](#_Toc146119027)

[5.9 Scoping rules 4](#_Toc146119028)

[6 Elementary data types 4](#_Toc146119029)

[6.1 Introduction 4](#_Toc146119030)

[6.2 Constant-length direct representation bit fields 5](#_Toc146119031)

[6.2.1 Alignment 5](#_Toc146119032)

[6.2.2 Type 5](#_Toc146119033)

[6.2.3 Length 5](#_Toc146119034)

[6.2.4 Value 6](#_Toc146119035)

[6.2.5 Constants 6](#_Toc146119036)

[6.2.6 Look-ahead 6](#_Toc146119037)

[6.3 Variable length direct representation bit fields 6](#_Toc146119038)

[6.4 Constant-length indirect representation bit fields 7](#_Toc146119039)

[6.5 Variable length indirect representation bit fields 8](#_Toc146119040)

[6.6 Variable length strings 9](#_Toc146119041)

[6.7 String literals 10](#_Toc146119042)

[7 Composite data types 10](#_Toc146119043)

[7.1 Classes 10](#_Toc146119044)

[7.2 Base and derived classes 10](#_Toc146119045)

[7.3 Abstract classes 11](#_Toc146119046)

[7.4 Expandable classes 12](#_Toc146119047)

[7.5 Polymorphism in class declaration 13](#_Toc146119048)

[7.5.1 General 13](#_Toc146119049)

[7.5.2 Polymorphism for abstract classes 14](#_Toc146119069)

[7.6 Parameter types 14](#_Toc146119071)

[7.7 Arrays 15](#_Toc146119072)

[7.8 Multi-dimensional arrays 15](#_Toc146119074)

[7.9 Partial arrays 16](#_Toc146119075)

[7.10 Implicit arrays 17](#_Toc146119076)

[8 Non-parsable variables 17](#_Toc146119078)

[9 Syntactic flow control 18](#_Toc146119079)

[9.1 Conditionals 18](#_Toc146119080)

[9.2 Loops 19](#_Toc146119081)

[Annex A (informative) SDL user guide 21](#_Toc146119082)

[A.1 Getting started 21](#_Toc146119083)

[A.2 Advanced concepts 25](#_Toc146119084)

[A.3 Common patterns 25](#_Toc146119085)

[A.4 Tooling 25](#_Toc146119086)

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 in ISO/IEC 14496 or ISO/IEC 23000. This mechanism is based on a Syntactic Description Language (SDL), documented here in the form of syntactic description rules. It leverages concepts defined in the C-like syntax used in ISO/IEC 11172–1:1993 and ISO/IEC 13818–1:2007 into a well-defined framework that lends itself to object-oriented data representations. SDL assumes an object-oriented underlying framework in which bitstream units consist of “classes.” This framework is based on the typing system of the C++ and Java programming languages. SDL extends the typing system by providing facilities for defining bitstream-level quantities, and how they should be parsed.

The elementary constructs are described first, followed by the composite syntactic constructs, and arithmetic and logical expressions. Finally, syntactic control flow and built-in functions are addressed. Syntactic flow control is needed to support context-sensitive data. Several examples are used to clarify the structure.

Initially defined in ISO/IEC 14496-1, this specification is backward-compatible compared to the SDL specified in ISO/IEC 14496-1. That is, a valid SDL definition based on ISO/IEC 14496-1 is also a valid definition based on this specification. However, the inverse is not true. The main additions of this specification are:

* Definition of string types and string literals
* Definition of the float type
* Explicit definition of allowed operators
* Declaration of variable in the first expression of a for statement

The International Organization for Standardization (ISO) draws attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

ISO takes no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO that he/she is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO. Information may be obtained from the patent database available at [www.iso.org/patents](http://www.iso.org/patents).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those in the patent database. ISO shall not be held responsible for identifying any or all such patent rights.

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

# Scope

This document specifies a syntactic language for describing the structure of binary data composed of number and characters elements represented in their binary forms.

# 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

FLC

Fixed Length Code

constant-length direct representation bit fields

3.2

SDL

Syntactic Description Language

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

3.3  
non-parsable variable

a variable whose value is undefined until the first assignment statement and whose scope is the current scope defined by enclosing curly brackets ({ and })

3.4  
parsable variable

a variable whose value is initialised at declaration by reading bits from the bitstream and whose scope is the current class. The number of bits to be read from the current position in the stream is indicated by an explicit length attribute or by a stopping condition

# Rule conventions

## 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*;…

# General syntax aspects

## Operators

Arithmetic and logical operators defined in the SDL are provided below in descending precedence order:

*a***++** postfix increment

*a***--** postfix decrement

**\*** 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

**.** class member access

## Whitespace

One or more space, tab or carriage return characters constitute a whitespace and serve to demarcate tokens in the SDL definition. Whitespace is required between tokens except for the following tokens where whitespace on either side of the token is optional:

* Primitive tokens
* Range expression i.e., “1..3”
* Arithmetic and logical operators i.e., a=3

## Built-in operators

The following built-in operators are defined.

Rule O.1: lengthof() Operator

lengthof(variable)

This operator returns the length, in bits, of the quantity contained in parentheses. The length is the number of bits that was most recently used to parse the quantity at hand. A return value of 0 means that no bits were parsed for this variable i.e., it is a non-parsable variable.

## Endianness

All quantities shall be represented in the bitstream with the most significant byte first, and the most significant bit first. Explicit references to endianness should be made in an SDL declaration where this is not the case.

## Representation of numbers

The values of signed integer variables shall be represented in the bitstream using the two’s-complement representation. The values of float variables shall be represented in the bitstream using the interchange format for binary floating-point numbers defined in IEEE Std 754-2019 for the corresponding declared bit depth. Floating point variables using bit depths not defined by IEEE Std 754-2019 shall not be used.

## 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.*

## Binary values

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

Rule S.3: Binary value

0bbinary\_value

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

EXAMPLE ⎯

0b0010.0101

## Hexadecimal values

To designate 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 value

0xhexadecimal\_value

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

NOTE 0x0F is equivalent to 0b0000.1111.

EXAMPLE ⎯

0xCAFE.BEEF

## Scoping rules

All parsable variables have class scope, i.e., they are available as class member variables.

For non-parsable variables, a new scope is introduced by the character '{' and exited by the character '}'. In particular, only variables declared in class scope are considered class member variables and are thus available in objects of that particular type.

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

# 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

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

### Alignment

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

For example, a 32‑bit value aligned on a byte boundary:

EXAMPLE ⎯

aligned bit(32) foo;

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, a 32‑bit value aligned on a 2‑byte boundary:

EXAMPLE ⎯

aligned(16) bit(32) foo;

The value of parsable variables with declarations that fall outside the flow of declarations shall be set to 0.

### 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 presence of the optional 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.

The optional length attribute is always present, except if the data is non-parsable i.e., it is not included in the bitstream.

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

EXAMPLE ⎯

unsigned int(5) parsable\_variable;

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

EXAMPLE ⎯

unsigned int non\_parsable\_variable;

### Value

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

### Constants

Constants shall be defined using the keyword const.

EXAMPLE ⎯

const int SOME\_VALUE=255; // non-parsable constant

const bit(3) BIT\_PATTERN=1; // this is equivalent to the bit string “001”

### 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 size parentheses to modify the parse size semantics.

Rule E.2: Look-ahead parsing

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

The keyword aligned and its modifier have the same definition 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 ⎯

aligned unsigned int (32)\* next\_code;

switch(next\_code == 100) {

case 100:

class\_100 foo;

default:

class\_default foo;

}

NOTE The control statement switch is defined in subclause 9.1.

## 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 ⎯

unsigned int(3) precision;

int(precision) DC;

## 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 MapName (output\_type) {

index**,** {value\_1**,** … value\_M},

…

}

The content of the map is defined 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 definition 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 Values. For the case of constant-length indirect representation bit fields, these Binary 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 defined **class** (classes are defined in 7.1).

Output values used to populate the output\_type shall be specified as aggregates surrounded by curly braces.

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 defined using the following representation:

EXAMPLE ⎯

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}

}

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

EXAMPLE ⎯

map offsets (int) {

0b00, {1},

0b01, {2},

0b10, {4}

}

The next rule describes the use of such a map.

Rule E.4: Mapped data types

output\_type(MapName) name**;**

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

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

EXAMPLE ⎯

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

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

unsigned int(8) u\_width;

unsigned int(8) u\_height;

}

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

EXAMPLE ⎯

int(offsets) index\_offset;

if(index\_offset == 2) {

unsigned int(6) foo;

}

## 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 ⎯

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 type declarations are allowed for map output values.

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

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 ⎯

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(32)},

0b0000.0000.0, {0, -20}

}

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.5: String data types

string\_type string\_name [**=** [**u**|**u8**]**"**string\_value**"**]**;**

The string\_type may be any of the following: utf8string, utfstring, utf8list, base64string. The format of those string types is defined in Table 1 —. In these definitions, 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.

1. String data type definitions

|  |  |
| --- | --- |
| **Name** | **Format** |
| 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 |

String data types can only be used for parsable variables.

EXAMPLE ⎯

utf8string message;

base64string encoded\_data;

## String literals

The *string*\_*value* attribute shall represent a string literal as a sequence of characters enclosed in double quotation marks (**"** and **"**) with an allowed encoding prefix. The encoding prefix is one of the following: u8 for UTF-8 string literal or ufor UTF-16 string literal.

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 ⎯

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";

# Composite data types

## Classes

Classes are the mechanism with which definitions of composite types is performed. Their definition is as follows.

Rule C.1: Classes

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

[element**;** …]

}

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

The different *element* declarations within the curly 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 variable declared in a class may be accessed using the dot (“**.**”) operator.

NOTE 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.

## 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 bitstream syntax declarations specified in the derived class.

Rule C.2: Derived classes

[aligned[(modifier)]] class class\_name [extends base\_class] [**:** bit(length) [id\_name **=**] object\_id | id\_range | *extended\_id\_range* ] {

[element**;** …]

}

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

The meaning of the keyword bit and its related attributes is defined in subclause 7.5.

NOTE Classes may also be encapsulated within other classes. In this case, the element in Rule C.2 is a class itself.

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

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

EXAMPLE ⎯

class foo {

int(3) a;

}

class bar extends foo {

int(5) b;

int(10) c;

}

## 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.3: Abstract classes

[aligned[(modifier)]] abstract class class\_name [extends base\_class] [**:** bit(length) [id\_name**=**] object\_id | id\_range | *extended\_id\_range* ] {

[element**;** …]

}

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

The meaning of the keyword bit and its related attributes defined in subclause 7.5.

EXAMPLE ⎯

abstract class Shape {

}

class Circle extends Shape {

unsigned int(8) radius;

}

class Rectangle extends Shape {

unsigned int(8) width;

unsigned int(8) height;

}

// only Circle and Rectangle classes can be present

class Example {

Circle c;

Rectangle r;

}

## 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.3: Expandable classes

[aligned[(modifier)]] expandable[(max\_class\_size)] class class\_name [extends base\_class] [: bit(length) [id\_name**=**] object\_id | id\_range | *extended\_id\_range* ] {

[element**;** …]

}

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

The meaning of the keyword bit and its related attributes defined in subclause 7.5.

Expandable classes may be used for classes that are required to support future compatible extensions or that may include private data. A legacy device can decode an expandable class up to the last parsable variable that has been defined for a given revision of this class and skip the unknown class data following the last known syntax element based on the class size information.

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 convention:

int sizeOfInstance = 0;

bit(1) nextByte;

bit(7) sizeByte;

sizeOfInstance = sizeByte;

while(nextByte) {

bit(1) nextByte;

bit(7) sizeByte;

sizeOfInstance = sizeOfInstance << 7 | sizeByte;

}

NOTE The encoding of the size information is per definition always byte-aligned.

An expandable class shall be defined in a way to ensure that its size is always an integer number of bytes. The size information is implicitly accessible within the class as the member variable sizeOfInstance whenever a class is made expandable.

If the class definition uses the **bit** keyword indicating encoding of a value representing an object\_id value, the encoding of this value shall precede the size encoding. The size information shall not include the number of bytes needed for the size encoding nor the *object\_id* value encoding.

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

If the expandable keyword has a maxClassSize attribute, then this indicates the maximum permissible size of this class in bytes, i.e. a maximum permissible value for sizeOfInstance. This information can help a parser to determine the appropriate type of integer to choose for holding the value of the variable sizeOfInstance.

Expandable class shall not derive from another expandable class.

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

EXAMPLE ⎯

aligned expandable(120) class Example {

int(3) a;

}

## Polymorphism in class declaration

### General

If the bit keyword is used, 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 the SDL syntax description. The object\_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 object\_id attribute value is given by the length attribute following the bit keyword. The optional attribute id\_name allows to access this object\_id value from within the class.

The actual class to be parsed is determined as follows:

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

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

EXAMPLE ⎯

class Foo : bit(2) id = 0 {

int(5) a;

}

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

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

}

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

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

}

class Example {

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

}

NOTE Derived classes may use the same object\_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.

As alternative to the object\_id attribute, it is also possible to have one of these attributes:

* 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 object\_id is specified as a comma-separated list of object\_id and range\_id 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 an object\_id or an id\_range or *extended\_id\_range*. This value or range of values shall correspond to legal values defined for the base class.

EXAMPLE ⎯

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

}

### 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 IDs available. For the abstract base class, the object\_id attribute shall be equal to the value 0.

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

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

}

## Parameter types

A parameter type defines a class with parameters. This addresses cases where the data structure of the class depends on variables of one or more other objects. As SDL follows a declarative approach, in such cases, references to other objects cannot be performed directly apart from within the body of a class (because objects are not instantiated). Parameter types provide placeholders for such references, in the same way as the arguments in a C function declaration. The syntax of a class definition with parameters is as follows.

Rule C.4: Class parameter types

[aligned[(modifier)]] [abstract] [expandable[(max\_class\_size)]] class class\_name [(parameter\_list)] [extends base\_class] [**:** bit(length) [id\_name**=**] object\_id | id\_range | *extended\_id\_range*] {

[element**;** …]

}

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

The *parameter\_list* is a list of type or **class** names and variable name pairs separated by commas. Any parsed or non-parsed variable value accessible within the current scope, can be passed as a parameter.

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

EXAMPLE ⎯

class A {

unsigned int(4) format;

}

class B (A a, int i) { // B uses parameter types

unsigned int(i) bar;

if( a.format == SOME\_FORMAT ) {

int(10) b;

}

}

class C {

int(2) i;

A a;

B foo(a, i); // parameters with populated values are required

}

## Arrays

Arrays are defined using square brackets. The array declaration is applicable to both elementary as well as composite objects.

Rule A.1: Arrays

*typespec* name [length]**;**

*typespec* is a type specification (e.g., an elementary type including bitstream representation information, e.g. **‘**int(2)**’**) or a class type. The attribute name is the name of the array, and length is its length. The length value can depend on run-time parameters such as other bitstream values or expressions that involve such values.

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 ⎯

unsigned int(4) a[5];

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

EXAMPLE ⎯

int(10) b;

int(2) c[b];

## Multi-dimensional arrays

Multi-dimensional arrays are supported as well. 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

typespec name [length][length]…**;**

In the following example, a is an array of 5 elements, each of which is represented as an array of 6 elements using 4 bits in the bitstream and interpreted as an unsigned integer:

EXAMPLE ⎯

unsigned int(4) a[5][6];

## 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 declaration is allowed in which individual elements of the array may be accessed.

Rule A.3: Partial arrays

typespec name[[index]]**;**

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. This notation is also valid for multidimensional arrays.

EXAMPLE ⎯

int a[4]; // a is a non-parsable variable which is an array of 4 elements

int(2) a[[0]]; // read first 2 bits for the 1st element of a

int(2) a[[1]]; // read 2 more bits for the 2nd element of a

int(2) a[[2]]; // read 2 more bits for the 3rd element of a

int(2) a[[3]]; // read 2 last bits for the 4th element of a

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

EXAMPLE ⎯

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

The following example indicates the entire 6th column (index ‘5’) of the array, with a length of 3 elements of 4-bit integer values:

EXAMPLE ⎯

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

The following example indicates the entire 4th row (index ‘3’) of the array, with a length of 5 elements of 4-bit integer values:

EXAMPLE ⎯

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

NOTE a[5] specifies an array of five elements, whereas a[[5]] specifies the 6th element of an array which can contain at least six elements.

## Implicit arrays

An array with an implicit length is an implicit array. It is indicated by an array declaration 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.4: Implicit arrays

typespec name [[range]];

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 defined, derived from the base class Foo (which may or may not be abstract):

EXAMPLE ⎯

class Foo : int(16) id = 0 {

int(5) a;

}

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

EXAMPLE ⎯

Foo explicit\_length\_array[100];

For an array of such objects, it is possible to implicitly determine the length by examining the validity of the object\_id of the class. Objects are inserted in the array as long as the object\_id can be properly resolved to one of the object\_id values defined in the base (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 object\_id. In this case, the SDL definition 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 object\_id resolution:

EXAMPLE ⎯

Foo f[]; // length implicitly obtained via object\_id resolution

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

EXAMPLE ⎯

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

# Non-parsable variables

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.

These are strictly of local scope to the class they are defined in. They may be used in expressions and conditions in the same way as bitstream-level variables. In the following example, the number of non-zero elements of an array is computed.

EXAMPLE ⎯

int(4) array[100];

int i; // this is a temporary, non-parsable variable

int n = 0; // this is a temporary, non-parsable variable

for (i=0; I < size; i++) {

if (array[[i]]!=0)

n++;

}

int(3) coefficients[n]; // read as many coefficients as there are non-zero elements in array

# 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. A condition equal to zero corresponds to false, and non-zero condition corresponds to true.

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

if (condition) {

…

} [ else if (condition) {

…

}] [else {

…

}]

In the following example, the presence of the entity ‘bar’ is determined by the ‘bar\_flag’:

EXAMPLE ⎯

class conditional\_object {

unsigned int(3) foo;

bit(1) bar\_flag;

if (bar\_flag) {

unsigned int(8) bar;

}

unsigned int(32) more\_foo;

}

NOTE The use of a flag necessitates its declaration before the conditional is encountered.

In the following example, two different representations for ‘bar’ are allowed, depending on the value of ‘bar\_flag’.

EXAMPLE ⎯

class conditional\_object {

unsigned int(3) foo;

bit(1) bar\_flag;

if (bar\_flag) {

unsigned int(8) bar;

} else {

unsigned int(16) bar;

}

unsigned int(32) more\_foo;

}

NOTE The use of a flag necessitates its declaration before the conditional is encountered.

We could equally well have another entity instead of the second version (16-bit) of ‘bar’. Also, if a variable appears twice (as in the example below), the declared ***type*** shall be identical.

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

Rule FC.2: Flow control using switch

switch (condition) {

[case label:  
 …   
 break;]

[default:…]

}

NOTE The case and the default statements may be optionally placed into ‘{’ and ‘}’.

EXAMPLE ⎯

unsigned int(32) type;

switch(type) {

case 0:

Foo f;

break;

case 0:

Bar b;

break;

default:

Def d;

}

## Loops

The same category of context-sensitive objects also includes iterative definitions of objects. These simply imply the repetitive use of the same syntax to parse the bitstream, until some condition is met (it is the conditional repetition that implies context, but fixed repetitions are obviously treated the same way). The familiar structures of ‘for’, ‘while’, and ‘do’ loops can be used for this purpose.

In the **for**-loop syntax, expression1, expression2 and expression3 each constitute a single statement. expression1 can be either a variable declaration with an assigned value or a value assignment and is executed prior to starting the repetitions. Then expression2 is evaluated, and if it is non-zero (true) the declarations within the braces are executed, followed by the execution of expression3. The process repeats until expression2 evaluates to zero (false).

Rule FC.3: Flow control using for

for (expression1; expression2; expression3) {

…

}

In the **do-while**-loop syntax, the block of statements is executed until condition evaluates to false.

NOTE The block will be executed at least once.

Rule FC.4: Flow control using do

do {

…

} while (condition);

In the **while**-loop syntax, the block is executed zero or more times, as long as condition evaluates to true.

Rule FC.5: Flow control using while

while (condition) {

…

}

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. In order to avoid confusion with standards written using the SDL syntax, we will take the example of the MPEG-2 transport packet and construct a possible SDL declaration to describe it.

According to ISO/IEC 13818-1, the binary structure of a transport packet is:

Table 2-2 – Transport packet of this Recommendation | International Standard

| 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 above, a transport packet is declared as a logical structure which can be encoded in a bitstream. A convenient way to reuse the same logical structure in SDL is to declare a **class**.

The minimum declaration for a class is a name 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 the Table 2-2, we can see that the first byte of a transport packet is the sync\_byte. Let’s then declare this first byte in our new class.

class transport\_packet {

unsigned int(8) sync\_byte;

}

We chose to declare our first variable as unsigned integer and call it sync\_byte. Since its length is one byte, we pass the number 8 in parenthesis as the length attribute of this variable. But we can go a step further. Since the ISO/IEC 13818-1 standard requires the sync\_byte to be equal to '0100 0111' (0x47), we can also declare that this variable must 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 in the transport packet which are transport\_error\_indicator, payload\_unit\_start\_indicator, and transport\_priority. SDL allows to have syntax elements of any raw binary data using the keyword **bit** which we will use. 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 declare 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 syntax elements whose values are defined as bit strings, e.g. '00' = Not scrambled, '01' = User-defined, etc. For simplicity, we will thus also define those variables as bit variables so that we can reuse directly the defined values in our declaration.

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 2-2, the reminder of the packet depends on the value of the adaptation\_field\_control element.

To represent this, we can use the **if**-statement defined by the SDL specification.

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 relation equal ‘**==**’ operator. Here since they are bit variables we use the binary string literal representation with the b-prefix. Finally, the logical combination of tests is achieved using the logical operator OR ‘**||**’.

All that remains to declare is the two bodies of the **if**-statement. In the first one, a logical structure adaptation\_field() is present. To represent this, let’s assume that we have previously defined a **class** adaptation\_field so that we declare 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 declare how large this data field is as long as the declaration of the class adaptation\_field determines it.

To complete the transport packet, we need to declare the body of the second **if**-statement. The ISO/IEC 13818-1 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 as well which would give:

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, just as if this SDL was defining executable code. If the SDL parser does not need those values this may be acceptable, however here this data constitutes the payload of the transport packet and we intend to keep this data intact. An elegant way to declare such sequence of bytes is by declaring an array of N-1 element instead of using a **for**-loop as follows:

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 transport\_packet class is almost over. There is still an unknown variable we are using which is N. 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 declare 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 as 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 ‘**.**’ operator and calculate the size of the array.

With this last variable, this completes the declaration of the MPEG-2 transport packet as an SDL object.

* 1. Advanced concepts

[Editor’s note: Some advanced concepts in more details.]

* + 1. **Byte alignment**
    2. **Constants**
    3. **Maps**
    4. **Class inheritance**
    5. **Abstract class**
    6. **Expandable class**
    7. **Parameter types**
    8. **Array**
  1. Common patterns

[Editor’s note: Discussion on top level scope/entry point via implicit array and using the following MPEG2 table as an example:]

[Editor’s note: Some example of patterns such as Type–length–value (TLV), e.g. ISOBMFF box definition based on 4ccs.]

* 1. Tooling

[Editor’s note: Description of tooling for validating SDL syntax? generating binary parser from SDL?, etc..]