**ISO 23001-19:20##(X)**

ISO TC 1/SC 29/WG 03

Date: 2022-07-22

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

WD 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 iv](#_Toc109403356)

[Introduction v](#_Toc109403357)

[1 Scope 1](#_Toc109403358)

[2 Normative references 1](#_Toc109403359)

[3 Terms and definitions 1](#_Toc109403360)

[4 Elementary data types 1](#_Toc109403361)

[4.1 Introduction 1](#_Toc109403362)

[4.2 Constant-Length Direct Representation Bit Fields 1](#_Toc109403363)

[4.3 Variable Length Direct Representation Bit Fields 2](#_Toc109403364)

[4.4 Constant-Length Indirect Representation Bit Fields 3](#_Toc109403365)

[4.5 Variable Length Indirect Representation Bit Fields 3](#_Toc109403366)

[4.6 Variable length string 4](#_Toc109403367)

[5 Composite Data Types 5](#_Toc109403368)

[5.1 Classes 5](#_Toc109403369)

[5.2 Abstract Classes 6](#_Toc109403370)

[5.3 Expandable classes 6](#_Toc109403371)

[5.4 Parameter types 7](#_Toc109403372)

[5.5 Arrays 8](#_Toc109403373)

[5.6 Partial Arrays 8](#_Toc109403374)

[5.7 Implicit Arrays 9](#_Toc109403375)

[6 Arithmetic and Logical Expressions 9](#_Toc109403376)

[7 Non-Parsable Variables 9](#_Toc109403377)

[8 Syntactic Flow Control 10](#_Toc109403378)

[9 Built-In Operators 11](#_Toc109403379)

[10 Scoping Rules 11](#_Toc109403380)

[Annex A (normative) SDL grammar 13](#_Toc109403381)

[A.1 General 13](#_Toc109403382)

[A.2 Grammar file 13](#_Toc109403383)

[Bibliography 15](#_Toc109403384)

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 a number of standard parts such as in ISO/IEC 14496 or ISO/IEC 23010. This mechanism is based on a Syntactic Description Language (SDL), documented here in the form of syntactic description rules. It directly extends 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. In particular, 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 take into account context-sensitive data. Several examples are used to clarify the structure.

[Editor’s note] Since ISO/IEC 14496-1 (from which this new part is extracted) lists a number patents in Annex J, the text below is kept according to the ISO template.

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 — MPEG systems technologies — Part 19: Syntactic description language

# Scope

Type text.

# Normative references

There are no normative references in this document.

# Terms and definitions

For the purposes of this document, the following terms 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

**Syntactic Description Language (SDL)**

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

# Elementary data types

## Introduction

The SDL uses the following elementary data types:

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.
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).
3. Constant-length indirect representation bit fields. These require an extra lookup into an appropriate table or variable to obtain the desired value or set of values.
4. Variable-length indirect representation bit fields (e.g., Huffman codes).

These elementary data types are described in more detail in the Clauses to follow immediately.

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

## Constant-Length Direct Representation Bit Fields

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

Rule E.1: Elementary Data Types

[aligned] type[(length)] element\_name [= value]; // C++-style comments allowed

The type may be any of the following: int for signed integer, unsigned int for unsigned integer, double for floating point, and bit for raw binary data. The length attribute indicates the length of the element in bits, as it is actually stored in the bitstream. Note that a data type equal to double shall only use 32 or 64 bit lengths. The value attribute shall be present only when the value is fixed (e.g., start codes or object IDs), and it may also indicate a range of values (i.e., ‘0x01..0xAF’). The type and the optional length attributes are always present, except if the data is non-parsable, i.e., it is not included in the bitstream. The keyword aligned indicates that the data is aligned on a byte boundary. As an example, a start code would be represented as:

aligned bit(32) picture\_start\_code=0x00000100;

An optional numeric modifier, as in aligned(32), may be used to signify alignment on other than byte boundary. Allowed values are 8, 16, 32, 64, and 128. Any skipped bits due to alignment shall have the value ‘0’. An entity such as temporal reference would be represented as:

unsigned int(5) temporal\_reference;

where unsigned int(5) indicates that the element shall be interpreted as a 5-bit unsigned integer. By default, data shall be represented with the most significant bit first, and the most significant byte first.

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

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”

To designate binary values, the 0b prefix shall be used, similar to the 0x prefix for hexadecimal numbers. A period (‘.’) may be optionally placed after every four digits for readability. Hence 0x0F is equivalent to 0b0000.1111.

In several instances, it may be desirable to examine the immediately following bits in the bitstream, without actually 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] type (length)\* element\_name;

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

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

## Variable Length Direct Representation Bit Fields

This case 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.

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 through the use of a table or map. In other words, the value extracted from the bitstream is an index to a table 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},

…

}

These tables are used to translate or map bits from the bitstream into a set of one or more values. The input type of a map (the index specified in the first column) shall always be bit. The output\_type entry shall be either a predefined type or a defined class (classes are defined in ‎8.3.1). The map is defined as a set of pairs of such indices and values. Keys are binary string constants while values are output\_type constants. Values shall be specified as aggregates surrounded by curly braces, similar to C or C++ structures.

EXAMPLE ⎯

class YUVblocks {// classes are fully defined later on

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}, // 4:2:0

0b01, {4, 2, 2}, // 4:2:2

0b10, {4, 4, 4} // 4:4:4

}

The next rule describes the use of such a map.

Rule E.4: Mapped Data Types

type (MapName) name;

The type of the variable shall be identical to the type returned from the map.

EXAMPLE ⎯

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

Using the above declaration, a particular value of the map may be accessed using the construct: chroma\_format.Ublocks.

## 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:

class val {

unsigned int foo;

int bar;

}

map sample\_vlc\_map (val) {

0b0000.001, {0, 5},

0b0000.0001, {1, -14}

}

The only difference is that the indices of the map are now of variable length. The variable-length codewords are (as before) binary strings, expressed by default in ‘0b’ or ‘0x’ format, optionally using the period (‘.’) every four digits for readability.

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

EXAMPLE ⎯ This example uses the class type ‘val’ as defined above.

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}

}

When the codeword 0b0000.0000.1 is encountered in the bitstream, then the value ‘5’ is assigned to the first element (val.foo). The following 32 bits are parsed and assigned as the value of the second element (val.bar). Note that, in case more than one element utilizes a parsable type declaration, the order is significant and is the order in which elements are parsed. In addition, the type within the map declaration shall match the type used in the class declaration associated with the map’s return type.

## Variable length string

Variable length string shall be represented as:

Rule E.5: String Data Types

string\_type string\_name [= prefix value]; // C++-style comments allowed

The string\_type may be any of the following: **utf8string**, **utfstring**, **utf8list**, **base64string**. The format of those string types is define in . 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.

1. String data type definitions

|  |  |
| --- | --- |
| **Name** | **Format** |
| **utf8string** | UTF-8 string as defined in IETF 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 base64 encoded data |

String data types can only be parsable data. The *value* attribute shall represent string literal as a sequence of characters enclosed in double quotation marks (" "). The *prefix* is one of the following: **u8** for UTF-8 string literal or **u** for UTF-16 string literal. When a string literal is present, the encoding format of the string literal needs to be allowed by the string data type of the variable.

EXAMPLE ⎯

utf8string message;

utfstring code = u8"this is a code";

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

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

base64string encoded\_data;

# Composite Data Types

## Classes

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

Rule C.1: Classes

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

[element; …] // zero or more elements

}

The different elements within the curly braces are the definitions of the elementary bitstream components discussed in 12.2 or control flow elements that will be discussed in a subsequent Subclause.

The optional keyword extends specifies that the class is “derived” from another class. Derivation implies that all information present in the base class is also present in the derived class, and that, in the bitstream, all such information *precedes* any additional bitstream syntax declarations specified in the new class.

The optional attribute id\_name allows to assign an object\_id, and, if present, is the key demultiplexing entity which allows differentiation between base and derived objects. It is also possible to have a range of possible values: the id\_range is specified as start\_id .. end\_id, inclusive of both bounds. It is also possible to have a combination of id\_range and object\_id: the *extended\_id\_range* is specified as a comma-separated list of object\_id and range\_id; for example, id\_name=object\_id1, object\_id2, start\_id .. end\_id.

If the attribute id\_name is used, a derived class may appear at any point in the bitstream where its base class is specified in the syntax. This allows to express polymorphism in the SDL syntax description. The actual class to be parsed is determined as follows:

* The base class declaration shall assign a constant value or range of values to object\_id.
* Each derived class declaration shall assign a constant value or ranges of values to object\_id. This value or set of values shall correspond to legal object\_id value(s) for the base class.

NOTE 1 — Derivation of classes is possible even when object\_ids are not used. However, in that case derived classes may not replace their base class in the bitstream.

NOTE 2 — Derived classes may use the same object\_id value as the base class. In that case classes can only be discriminated through context information.

EXAMPLE ⎯

class slice: aligned bit(32) slice\_start\_code=0x00000101 .. 0x000001AF {

// here we get vertical\_size\_extension, if present

if (scalable\_mode==DATA\_PARTITIONING) {

unsigned int(7) priority\_breakpoint;

}

…

}

class foo {

int(3) a;

...

}

class bar extends foo {

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

int(10) c;

...

}

The order of declaration of the bitstream components is important: it is the same order in which the elements appear in the bitstream. In the above examples, bar.b immediately precedes bar.c in the bitstream.

Objects may also be encapsulated within other objects. In this case, the element in Rule C.1 is an object itself.

## Abstract Classes

When the abstract keyword is used in the class declaration, it indicates that only derived classes of this class shall be present in the bitstream. This implies that the derived classes may use the entire range of IDs available. The declaration of the abstract class requires a declaration of an ID, with 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 {

...

}

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

...

}

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". In this case the class encodes its own size in bytes explicitly. This may be used for classes that require future compatible extension or that may include private data. A legacy device is able to decode an expandable class up to the last parsable variable that has been defined for a given revision of this class. Using the size information, the parser shall skip the class data following the last known syntax element. Anywhere in the syntax where a set of expandable classes with object\_id is expected it is permissible to intersperse expandable classes with unknown object\_id values. These classes shall be skipped, using the size information.

The size encoding precedes any parsable variables of the class. If the class has an object\_id, the encoding of the object\_id precedes the size encoding. The size information shall not include the number of bytes needed for the size and the object\_id encoding. Instances of expandable classes shall always have a size corresponding to an integer number of bytes. The size information is accessible within the class as class instance variable sizeOfInstance.

If the expandable keyword has a maxClassSize attribute, then this indicates the maximum permissible size of this class in bytes, including any expansion.

The length encoding is itself defined in SDL as follows:

int sizeOfInstance = 0;

bit(1) nextByte;

bit(7) sizeOfInstance;

while(nextByte) {

bit(1) nextByte;

bit(7) sizeByte;

sizeOfInstance = sizeOfInstance<<7 | sizeByte;

}

## Parameter types

A parameter type defines a class with parameters. This is to address cases where the data structure of the class depends on variables of one or more other objects. Since SDL follows a declarative approach, references to other objects, in such cases, cannot be performed directly (none is 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.2: Class Parameter Types

[aligned] [abstract] class object\_name [(parameter list)] [extends parent\_class]

[: bit(length) [id\_name=] object\_id | id\_range ] {

[element; …] // zero or more elements

}

The parameter list is a list of type names and variable name pairs separated by commas. Any element of the bitstream, or value derived from the bitstream with a variable-length codeword, or a constant, can be passed as a parameter.

A class that uses parameter types is dependent on the objects in its parameter list, whether class objects or simple variables. When instantiating such a class into an object, the parameters have to be instantiated objects of their corresponding classes or types.

EXAMPLE ⎯

class A {

// class body

...

unsigned int(4) format;

}

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

unsigned int(i) bar;

...

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

...

}

...

}

class C {

int(2) i;

A a;

B foo( a, I); // instantiated parameters are required

}

## Arrays

Arrays are defined in a similar way as in C/C++, i.e., using square brackets. Their length, however, can depend on run-time parameters such as other bitstream values or expressions that involve such values. 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 (including bitstream representation information, e.g. ‘int(2)’). The attribute name is the name of the array, and length is its length.

EXAMPLE ⎯

unsigned int(4) a[5];

int(10) b;

int(2) c[b];

Here ‘a’ is an array of 5 elements, each of which is represented using 4 bits in the bitstream and interpreted as an unsigned integer. In the case of ‘c’, its length depends on the actual value of ‘b’. Multi-dimensional arrays are allowed 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 .

## Partial Arrays

In several situations, it is desirable to load the values of an array one by one, in order to check, for example, 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.2: Partial Arrays

typespec name[[index]];

Here index is the element of the array that is defined. Several such partial definitions may be given, but they shall all agree on the type specification. This notation is also valid for multidimensional arrays.

EXAMPLE ⎯

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

indicates the element a(5, 3) of the array (the element in the 6th row and the 4th column), while

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

indicates the entire sixth column of the array, and

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

indicates the entire fourth row of the array, with a length of 5 elements.

NOTE ⎯ **a[***5***]** means that the array has five elements, whereas **a[[***5***]]** implies that there are at least six.

## Implicit Arrays

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. Let us assume that a set of polymorphic classes is defined, derived from the base class Foo (may or may not be abstract):

class Foo : int(16) id = 0 {

...

}

For an array of such objects, it is possible to implicitly determine the length by examining the validity of the class ID. Objects are inserted in the array as long as the ID can be properly resolved to one of the IDs defined in the base (if not abstract) or its derived classes. This behavior is indicated by an array declaration without a length specification.

EXAMPLE 1 ⎯

class Example {

Foo f[]; // length implicitly obtained via ID resolution

}

To limit the minimum and maximum length of the array, a range specification may be inserted in the specification of the length.

EXAMPLE 2 ⎯

class Example {

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

}

In this example, ‘f’ may have at least 1 and at most 255 elements.

# Arithmetic and Logical Expressions

All standard arithmetic and logical operators of C++ are allowed, including their precedence rules.

# Non-Parsable Variables

In order to accommodate complex syntactic constructs, in which context information cannot be directly obtained from the bitstream but only as a result of a non-trivial 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.

unsigned int(6) size;

int(4) array[size];

…

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

for (i=0, n=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

The syntactic flow control provides constructs that allow conditional parsing, depending on context, as well as repetitive parsing. The familiar C/C++ if-then-else construct is used for testing conditions. Similarly to C/C++, zero corresponds to false, and non-zero corresponds to true.

Rule FC.1: Flow Control Using If-Then-Else

if (condition) {

…

} [ else if (condition) {

…

}] [else {

…

}]

EXAMPLE 1 ⎯

class conditional\_object {

unsigned int(3) foo;

bit(1) bar\_flag;

if (bar\_flag) {

unsigned int(8) bar;

}

unsigned int(32) more\_foo;

}

Here the presence of the entity ‘bar’ is determined by the ‘bar\_flag’.

EXAMPLE 2 ⎯

class conditional\_object {

unsigned int(3) foo;

bit(1) bar\_flag;

if (bar\_flag) {

unsigned int(8) bar;

} else {

unsigned int(some\_vlc\_table) bar;

}

unsigned int(32) more\_foo;

}

Here we allow two different representations for ‘bar’, depending on the value of ‘bar\_flag’. We could equally well have another entity instead of the second version (the variable length one) of ‘bar’ (another object, or another variable). Note that the use of a flag necessitates its declaration before the conditional is encountered. Also, if a variable appears twice (as in the example above), the types shall be identical.

In order to facilitate cascades of if-then-else constructs, the ‘switch’ statement is also allowed.

Rule FC.2: Flow Control Using Switch

switch (condition) {

[case label1: …]

[default:]

}

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.

Rule FC.3: Flow Control Using For

for (expression1; expression2; expression3) {

…

}

expression1 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).

Note that it is allowed to include a variable declaration in expression1.

Rule FC.4: Flow Control Using Do

do {

…

} while (condition);

Here the block of statements is executed until condition evaluates to false. Note that the block will be executed at least once.

Rule FC.5: Flow Control Using While

while (condition) {

…

}

The block is executed zero or more times, as long as condition evalutes to non-zero (true).

# 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.

# Scoping Rules

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

For non-parsable variables, the usual C++/Java scoping rules are followed (a new scope is introduced by curly braces: ‘{‘ and ‘}’). In particular, only variables declared in class scope are considered class member variables, and are thus available in objects of that particular type.

1. (normative)  
     
   SDL grammar
   1. General

This clause provides the SDL grammar implementing the rules of the present specification. The grammar is based on the parsing expression grammar (PEG) concept and following the syntax and set of rules defined by the Pegen software project [1].

[Editor’s note] The following is in work-in-progress and may not reflect yet all the rules described in this specification.

* 1. Grammar file

1. SDL grammar file

|  |
| --- |
| start: file\_input  file\_input: (NEWLINE+ | line)\* ENDMARKER  line: class\_def | comment\_cpp  #NOTE: 14496-1 forbids going back to a new line berfore {  class\_def: aligned? abstract? 'class' NAME+ parameter\_list? parent\_class? NEWLINE? '{' body? '}'  aligned: 'aligned' '(' NUMBER+ ')'  abstract: 'abstract'  parameter\_list: '(' ','.parameter+ ')'  # NOTE: we allow arrays to be passed as parameter class, however 14496-1 is not clear on this  parameter: optional? type NAME array\_length?  # NOTE: Not in 14496-1, not sure where this come from  optional: 'optional'  # NOTE: would allow "unsigned bit", to be improved  type: signed? data\_type  signed: 'unsigned'  data\_type:  | 'bit'  | 'int'  | 'double'  array\_length: '[' NUMBER\* ']'  #NOTE: 14496-1 does not allow paramters after parent class name  parent\_class: 'extends' NAME '(' ','.value+ ')'  body: stmt\*  stmt:  | elementary\_data\_type  | non\_parsable\_variable  | object\_instantiation  | if\_stmt  | for\_stmt  | comment\_cpp  #TODO: See how to do any character up to newline  comment\_cpp: '//' (NAME | 'floor' | 'class' | 'if' | 'else' | 'for' | 'extends' | NUMBER | '==' | '=' | '{' | ';' | ',' | '-' | '/' | ':' | '?' )\*  elementary\_data\_type: template? aligned? const? type length NAME array\_length? assigned\_value? ';'  non\_parsable\_variable: template? const? type NAME array\_length? assigned\_value? ';'  #NOTE: Not in 14496-1 but used in 14496-12  template: 'template'  const: 'const'  length: '(' (NUMBER | NAME) ')'  #NOTE: array initialisation with {val1, val2, ...} not in 14496-1  assigned\_value: '=' (value | array\_initialisation)  object\_instantiation: NAME NAME ( '(' ','.value+ ')' )\* array\_length? ';'  #TODO: This rule should not allow whitepaces between name and '+'s  variable\_incr: NAME '+' '+'  #NOTE: STRING literal e.g. 'uuid' is not allowed in 14496-1  #NOTE: NUMBER catches decimal, octal, hexadecimal, binary, foating point (scientific noation) and even imaginary number. Too broad for SDL.  value: function | expr | NUMBER | NAME | STRING  expr: ( '(' value operator value ')' ) | (value operator value)  #NOTE: & and && not in 14496-1 but used in 14496-12  operator: operator\_test | operator\_bin | operator\_math  operator\_math: '+' | '-' | '/' | '\*'  operator\_test: '==' | '<=' | '<' | '>=' | '>' | '!='  operator\_bin: '&'  function: function\_name '(' value ')'  #NOTE: Only lengthof in 14496-1, floor is used in 14496-12 without definition  function\_name: 'floor' | 'lengthof'  array\_initialisation: '{' ','.value+ '}'  if\_stmt: 'if' '(' condition ')' '{' body '}' else\_if\_stmt? else\_stmt?  else\_if\_stmt: 'else' 'if' '(' condition ')' '{' body '}'  else\_stmt: 'else' '{' body '}'  condition: value  for\_stmt: 'for' '(' expression1 ';' expression2? ';' expression3 ')' '{' body '}'  expression1: variable\_assignment  variable\_assignment: NAME assigned\_value  expression2: value  expression3: variable\_incr |

Bibliography

1. Pegen documentation, <https://we-like-parsers.github.io/pegen/>