ISO/IEC JTC 1/SC 29/WG 03 N1653

**ISO/IEC JTC 1/SC 29/WG 03  
MPEG Systems   
Convenorship: KATS (Korea, Republic of)**

**Document type:** Output Document

**Title:** Technology under Consideration on ISO/IEC 14496-34 Syntactic Description Language

**Status:** Approved

**Date of document:** 2025-10-12

**Source:** ISO/IEC JTC 1/SC 29/WG 03

**No. of pages:** 16 (with cover page)

**Email of Convenor:** young.L @ samsung . com

**Committee URL:** <https://isotc.iso.org/livelink/livelink/open/jtc1sc29wg3>

**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 03 MPEG SYSTEMS**

**ISO/IEC JTC 1/SC 29/WG 03 N1653**

**October 2025, Geneva, CH**

|  |  |
| --- | --- |
| **Title** | **Technology under Consideration on ISO/IEC 14496-34 Syntactic Description Language** |
| **Source** | **WG 03, MPEG Systems** |
| **Status** | **Approved** |
| **Serial Number** | **25627** |

# Introduction

This document collects technologies being under study for consideration in the development of the standard ISO/IEC 14496-34 Syntactic Description Language

# Table of content

[1. Introduction 1](#_Toc211163205)

[2. Table of content 1](#_Toc211163206)

[3. Summary of topics 2](#_Toc211163207)

[4. Topic #1: Grammar file for SDL syntax 2](#_Toc211163208)

[5. Topic #2: On the keyword template 5](#_Toc211163209)

[6. Topic #3: On optional class member 6](#_Toc211163210)

[7. Topic #4: Array of boxes for ISOBMFF 7](#_Toc211163211)

[8. Topic #5: An unordered set of boxes 10](#_Toc211163212)

[9. Topic #6: Import directive 12](#_Toc211163213)

[10. Topic #7: SDL source file definition 12](#_Toc211163214)

[11. Topic #8: Unicode inconsistencies 13](#_Toc211163215)

[12. Topic #9: Improve the flexibility of class\_id 15](#_Toc211163216)

[13. Topic #10: Extra built-in operator 16](#_Toc211163217)

[14. Topic #11: Flexible class size encoding 16](#_Toc211163218)

[15. Topic #12: Implicit end of parsing 16](#_Toc211163219)

[16. Topic #13: Entry point 17](#_Toc211163220)

[17. Topic #14: Unexpected trailing bits 18](#_Toc211163221)

[18. Topic #15: Explicit parsing behaviour 18](#_Toc211163222)

[19. Topic #16: Extended value constraints 18](#_Toc211163223)

[20. Topic #17: Usage Phases concept 19](#_Toc211163224)

# Summary of topics

|  |  |  |  |
| --- | --- | --- | --- |
| #id | Name | Description | Source |
| 1 | Grammar file | An complete grammar file describing SDL syntax. | [m62029](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/6) |
| 2 | Keyword template | Creating a new keyword inspired by the template keyword in ISOBMFF. | [m67722](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/93) |
| 3 | Optional class member | Allowing members of classes to be optional, also optional class parameters. | [m69324](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/102) |
| 4 | Array of boxes | Enabling this syntax Box boxes[]; | [m69324](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/102) |
| 5 | Unordered set of boxes | A sequence of box wherein each box is optional but appears at most once in any order. | [m69324](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/102) |
| 6 | “Imported” SDL class | Possible directive to import other SDL files into a current SDL file. | [m62014](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/4) |
| 7 | SDL file definition | A standardised SDL file format to write SDL classes into a plain text file. | [m62015](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/5) |
| 8 | Unicode inconsistencies | Inconsistencies regarding Unicode handling. | [m71608](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/108) |
| 9 | Better class\_id | Improve the flexibility of class\_id. | [m73989](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/119) |
| 10 | Extra built-in operator | Add new built-in operators (similar to lengthof()) which assist with control flow parsing. | [m73989](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/119) |
| 11 | Flexible class size encoding | Support arbitrary encoding and location of class size, support arbitrary name, support arbitrary class size boundary and support length in bits. | [m73989](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/119) |
| 12 | Implicit end of parsing | Improve pre-defined parsing behaviours for variable length arrays, optional class members and “parse until” semantics. | [m73989](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/119) |
| 13 | Entry point | Explicitly state that an entry point much be a class. | [m73989](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/119) |
| 14 | Unexpected trailing bits | Explicitly state parsed behaviour for unexpected trailing bits. | [m73989](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/119) |
| 15 | Explicit parsing behaviour | Ability to remove parsing behaviour defined in the ISOBMFF standard descriptive text. | [m73989](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/119) |
| 16 | Extended value constraints | Allow definition of value constraints. | [m73989](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/119) |
| 17 | Parsing phases | Introducing parsing context concept via the introduction of a usage phases concept. | [m73968](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/120) |
| 18 | const for parsed variables | Remove support for the const keyword for parsed variables and define them as always immutable | [m73972](https://git.mpeg.expert/MPEG/Systems/sdl/contributions/-/issues/123) |

# Topic #1: Grammar file for SDL syntax

## 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].

Currently, the grammar file is hosted an developed at <http://mpegx.int-evry.fr/software/MPEG/Systems/sdl/sdl-grammar>.

The usage of PEG seems adequate for the validation of the SDL syntax, however there are still some questions whether the PEG grammar defined by the Pegen software project is appropriate for the developing the conformance of the SDL specification. Further study is encourage on other possible alternative for such grammar file.

## Grammar file

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

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 {  # Rule C.1 and C.2  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  | assignment\_stmt  | object\_instantiation  | increment\_stmt  | if\_stmt  | switch\_stmt  | for\_stmt  | do\_stmt  | while\_stmt  | comment\_cpp  #TODO: See how to do any character up to newline  comment\_cpp: '//' (NAME | 'floor' | 'class' | 'if' | 'else' | 'for' | 'extends' | NUMBER | '==' | '=' | '{' | ';' | ',' | '-' | '/' | ':' | '?' )\*  # Rule E.1 and A.1  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? ';'  variable\_assignment: NAME assigned\_value  assignment\_stmt: variable\_assignment ';'  #TODO: This rule should not allow whitepaces between name and '+'s  variable\_incr: NAME '+' '+'  increment\_stmt: variable\_incr ';'  #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\_logical | operator\_bin | operator\_math  operator\_math: '+' | '-' | '/' | '\*'  operator\_test: '==' | '<=' | '<' | '>=' | '>' | '!='  operator\_bin: '&' | '|'  operator\_logical: '&' '&' | '|' '|'  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+ '}'  # Rule FC.1  if\_stmt: 'if' '(' condition ')' '{' body '}' else\_if\_stmt? else\_stmt?  else\_if\_stmt: 'else' 'if' '(' condition ')' '{' body '}'  else\_stmt: 'else' '{' body '}'  condition: value  # Rule FC.2  switch\_stmt: 'switch' '(' condition ')' '{' (switch\_case switch\_break?)\* switch\_default? switch\_break?'}'  switch\_break: 'break' ';'  switch\_case: 'case' (NUMBER | NAME | STRING) ':' body?  switch\_default: 'default' ':' body?  # Rule FC.3  for\_stmt: 'for' '(' expression1 ';' expression2 ';' expression3 ')' '{' body '}'  for\_variable\_declaration\_assignment: type NAME array\_length? assigned\_value  expression1:  | variable\_assignment  | for\_variable\_declaration\_assignment  expression2: value  expression3: variable\_incr  # Rule FC.4  do\_stmt: 'do' '{' body '}' 'while' '(' condition ')' ';'  # Rule FC.5  while\_stmt: 'while' '(' condition ')' '{' body '}' |

## References

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

# Topic #2: On the keyword template

The template keywork in defined in ISO/IEC 14496-12 and does not belong to the original nor currently developped SDL.

The goal of the keyword is to allow other possible values than the one defined for the field by the assignment operator.

EXAMPLE ⎯

template int(32) rate = 0x00010000; // typically 1.0

In this example, the field rate shall be “0x00010000” for a file complying to this specification. But the keyword template allows a derivate spec to define another value.

From the point of a view of a file parser that only knows about the current specification, it shall throw an error is rate as a different value than “0x00010000”.

The ongoing 8ed of ISOBMFF is attempting to clarify the definition and the reader/writer behaviour.

It should be studied whether template should remain an ISOBMFF extension of the SDL or become a feature of the SDL.

# Topic #3: On optional class member

## General

Another typical case is when a box declares several fields followed some optional boxes. The meta box is also an example of that:

aligned(8) class MetaBox (handler\_type)  
 extends FullBox('meta', version = 0, 0)   
{  
 HandlerBox(handler\_type) theHandler; // optional  
 PrimaryItemBox primary\_resource[0..1]; // optional  
 DataInformationBox file\_locations; // optional  
 ItemLocationBox item\_locations; // optional  
 ItemProtectionBox protections; // optional  
 ItemInfoBox item\_infos; // optional  
 IPMPControlBox IPMP\_control; // optional  
 ItemReferenceBox item\_refs; // optional  
 ItemPropertiesBox item\_properties; // optional  
 ItemDataBox item\_data; // optional  
 GroupsListBox entity\_groups; // optional  
 Box other\_boxes[]; // optional  
}

## Discussion

It is possible to declare optional boxes in the ISOBMFF because:

1. Boxes can be disambiguated using the box\_type field in the header.
2. The box size in the box header allows the parser to determine if the end of the box is reached. If not, then this means that some additional boxes are present.

## Possible solutions

## Optional class members

In SDL, it can happen that a class may contain a given class based on the context it is in. In some environment, this class will be needed in some other this class would not be needed. As a result, the SDL author may want to declare a nested class instance to be optional.

For example:

class Foo  
{  
 unsigned int(8) a;  
 optional Bar bar;  
}

class Log

{

Foo foo;

}

In this class MyClass, the body of Foo does declare bar and but if bar is not in the parsed data, then the member foo.bar does not exist and thus would end up in an unspecified behaviour.

## Optional keyword

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

Rule C.2: Class data types

[optional] class\_identifier *class\_variable\_*identifier**;**

The keyword optional indicates that the class instance may or may not be present. To use this keyword, the class shall be defined with a class identifier.

[Editor’s note: member class mark as typeid? Or should we say that class shall be a derived class of base class for which the class identifier is defined].

Accessing an optional member of class that was not parsed, and thus does not exist in the class instance, result in an unspecified behaviour. Consequently, the value of this class are also unspecified.

NOTE The SDL specification does not provide a mean to check whether the member of class exists.

# Topic #4: Array of boxes for ISOBMFF

## On container box

In this scenario, a box is meant to be a container for other boxes. For example, the movie box is defined as follows:

**Movie box**

**Definition**

Box Type: 'moov'  
Container: File  
Mandatory: Yes  
Quantity: Exactly one

The structure-data for a presentation is stored in the single MovieBox which occurs at the top-level of a file. Normally this box is close to the beginning or end of the file, though this is not required.

**Syntax**

aligned(8) class MovieBox extends Box('moov')  
{  
}

From the SDL declaration of the movie box, nothing is specifying what the box can contain. The semantic is also silent on what it contains. Instead, the philosophy of the ISOBMFF specification is to specify where a box can be located and not what it can contain.

Therefore, strictly speaking, the movie box as declared is an empty box.

Another example of a container box is the MetaBox. In this case, specification declare an array of element of the class Box.

aligned(8) class MetaBox (handler\_type)  
 extends FullBox('meta', version = 0, 0)   
{  
 …  
 Box other\_boxes[]; // optional  
}

Note that the comment mentions optional which would imply that the array may be empty which is currently a topic of discussion in the File Format group (regarding the element track\_IDs[])

More generally, this syntax of declaring a generic class would be possible is the class Box would be an abstract class and that all derive classes would use the SDL mechanism of extension ID. However, those conditions are not met the current ISOBMFF specification.

## Discussion

In ISOBMFF, there are several places with a box can contains any boxes.

aligned(8) class MetaBox (handler\_type)  
 extends FullBox('meta', version = 0, 0)   
{  
 …  
 Box other\_boxes[]; // optional  
}

The statement “Box other\_boxes[]” is actually possible in SDL but ISOBMFF doesn’t currently use the SDL polymorphism feature (class identifier in the class declaration) to use this feature.

There seems to be three cases of container box:

1. A box can contain any other boxes.
2. A box can contain zero or more boxes from a any boxes in any order.
3. A box can contain zero or more boxes from a predetermined list of boxes in a specific order.

For case 1), the SDL has some provision but requires using specific tools which limits the freedom of the SDL declaration author.

For case 2), the SDL does not seem to have any tools for that.

For case 3), this is almost enabled in SDL as illustrated in the meta box. The only missing feature is to declare that the boxes are optional.

## Possible solutions

## New keyword “class id”

Declaring a class identifier keyword to declare which class member in the body should be used as class identifier.

Possible keywords are:

* **typeid**
* **id**
* **class\_id**

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 Bar {  
 id bit(2) class\_type = 0;  
 int(5) a; // this a is preceded by the 2 bits of id

}

Those two classes Foo and Bar would be syntactically equivalent.

To extend from it, the regular derivation process can be used but the SDL author has to make sure the same class id is defined in each derived class and at the same bit position in the class.

class Foo1 extends Foo {

id bit(2) class\_type = 1;

int(3) b;

}

class Foo2 extends Foo {  
 id bit(2) class\_type = 2;

int(5) c;

}

## Extending polymorphism declaration

Currently, the polymorphism feature works by defining a field bit which will be written as the first element of the class.

The issue with this in ISOBMFF, and possibly other specifications, is that the box type is written in the second position after the size of the box.

Therefore, the current polymorphism could be extended to allow for declaring the bit offset from which to start reading the class id.

For example, one way would be to add an optional bit offset, when absent the offset is null.

class FooBox : 32+bit(32) type = 'moov' {

unsigned int(32) size;  
 bit(32) type;  
 if (size==1) {  
 unsigned int(64) largesize;  
 } else if (size==0) {  
 // box extends to end of file  
 }  
 if (boxtype=='uuid') {  
 unsigned int(8) usertype[16] = extended\_type;  
 }

…

}

Alternatively, we could also forbid the redeclaration in the class.

class FooBox : 32+bit(32) type = 'moov' {

unsigned int(32) size;  
 // Here is the bit(32) type field but it is already declared in the class declaration   
 if (size==1) {  
 unsigned int(64) largesize;  
 } else if (size==0) {  
 // box extends to end of file  
 }  
 if (boxtype=='uuid') {  
 unsigned int(8) usertype[16] = extended\_type;  
 }

…

}

# Topic #5: An unordered set of boxes

## General

Here the case is that a box contain a sequence of box wherein each box is optional. However, the order of which the boxes appear is not specified and any order is allowed.

We can call this case an unordered set of boxes where set is understood as the concept of unordered set as in C++:

std::unordered\_set is an associative container that contains a set of unique objects of type Key.

So those solutions provide mean to express this type of data.

In spirit, it is also similar to the XML element all:

From 3.8.4.1.3 All-groups , <https://www.w3.org/TR/xmlschema11-1/>

[Definition:]  A **grouping** of a sequence is a set of sub-sequences, some or all of which may be empty, such that each member of the original sequence appears once and only once in one of the sub-sequences and all members of all sub-sequences are in the original sequence.

## Possible solutions

## New syntax for “an unordered set / all XML element”

Let’s take this box as example and assume the order of the clap and pasp boxes can be any.

class VisualSampleEntry(codingname) extends SampleEntry (codingname)  
{  
 unsigned int(16) pre\_defined = 0;  
..   
 CleanApertureBox clap; // optional  
 PixelAspectRatioBox pasp; // optional  
}

One way of expressing the unordered set it.

class UnorderedBox {

all {

   Second b;

    First a;

  }

}

One parsing can lead to:

First

Second

Another parsing can lead to:

Second

First

To be more explicit, the syntax could also incorporate the way to disambiguate the set, for example:

class UnorderedBox {

all : bit(32) id {

   Second b;

    First a;

  }

}

This mean that the parser has to read id as the next 32 bits to identify it correspond to b.id or a.id.

Another alternative is to declare the base class before the unordered set.

class UnorderedBox {

all : Box {

   FooBox b;

    BarBox a;

  }

}

In this case, FooBox and BarBox derive from Box and per their definition, the parser knows how to disambiguate between a and b.

# Topic #6: Import directive

## Motivation

We can envision two ways to reuse SDL definitions from other files, similar to include foo.xxx and import foo behaviour commonly used in programming language.

With include, the targeted file is simply replaced by the directive in the current file. The feature is simple to interpret. However, it typically requires to handle duplicated declarations when the same file is included multiple files, e.g. #define guard in C/C++.

Alternatively, the import functionality typically refer to importing the function of loading the declaration found into the targeted file in the current scope of the document. It thus does not imply the replacement by the targeted file itself.

As a result, it appears less to define an import feature instead of an include function which has to come with guard features.

## Text proposal

**Directives**

The following directives are defined.

Rule D.1: import

import path/to/module

This directive imports the SDL declarations found in the file called module.sdl at the located at the path “path/to”. After the import, all the classes defined in this imported module can be used in the subsequent SDL declarations.

# Topic #7: SDL source file definition

A SDL file is a plain text file with extension .sdl which contains a set of class object declarations and comment lines. Comment lines are defined as in 4.2 Comments. The character encoding could also be considered.

The Annex A which contains the SDL grammar is actually already anticipating a file containing the SDL declaration.

# Topic #8: Unicode inconsistencies

## Motivation

Clause 5.1 permits use of UCS characters "except" (a different way of saying "shall not"?) certain characters:

-          "the BOM character (0xFEFF)"

-          "non-printable characters (0x00 to 0x1F inclusive)"

-          "characters causing line feeds (0x85, 0x2028, 0x2029"

## Comment #1

First, a comment about citation of UCS characters: the notation "0x" is a convention from C or similar specifications for referring to hexadecimal integer values in a machine representation. It is not a convention used in ISO/IEC 10646 to refer to characters or code points. Related, it's important to note that UCS code points are integer values in the range 0000 to 10FFFF and conceptually distinct from any machine representation. When referring to 10646, the conventions of 10646 should be used. The relevant conventions are defined in clause 7.6, "Short identifiers for code points (UIDs)". Quoting a summary statement,

The full syntax of the notation of a short identifier, in Backus-Naur form, is   
                { U | u } {+}(xxxx | xxxxx | xxxxxx)

where "x" represents one hexadecimal digit (0 to 9, A to F, or a to f).

Thus "0x00 to 0x1F" should be cited as "U+0000 to U+001F"; "0x85, 0x2028, 0x2029" as "U+0085, U+2028, U+2029". (The "U+" is optional, but in this document should be included for clarity.)

Note: on this point, this document could do well to follow its own requirement for string literals in 5.18:

"A universal character code point shall be defined using 4 uppercase hexadecimal characters prefixed with **\u** or 8 uppercase hexadecimal characters prefixed with **\U**"

Also, btw, the distinction I mention between code points and machine representations is very important in 10646 and Unicode. See [UTR #17](https://www.unicode.org/reports/tr17/) and [RFC2130](https://www.ietf.org/rfc/rfc2130.txt) for background. I'll touch on this again below.

## Comment #2

10646 does not anywhere refer to "BOM". The character assigned to code point U+FEFF is named ZERO WIDTH NO-BREAK SPACE. The notion of a byte order "signature" is described in clause 7.3.7, but the terminology "BOM" or "byte order mark" is not used in 10646; rather, it comes from The Unicode Standard (see [Unicode 16.0, 2.13.2](https://www.unicode.org/versions/Unicode16.0.0/core-spec/chapter-2/#G9354)).

## Comment #3

Clause 5.1 states that an SDL specification cannot contain "BOM", but 6.6 states contexts in which BOM will be required. IIUC, this all seems to imply there can be cases in which the SDL description would need to specify a string literal that does include BOM, and by 5.1 and 5.18 it would need to be cited within a string literal using an escaped representation such as u"\ufeff". That seems to be worth mentioning in 5.1 in a note.

## Comment #4

Re this statement in 5.1:

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.

This statement seems to be confused about the meaning of "character set". As noted in [RFC2130](https://www.ietf.org/rfc/rfc2130.txt), "character set" has been used with different meanings in different contexts. That document uses "Coded Character Set" unambiguously in a manner consistent with ISO/IEC 10646 and Unicode. This document should do likewise.

Since this document is referencing 10646, then "character set" should be understood in the nearest terminology of that standard, namely "Coded Character Set", in which case it is incorrect to state that this document doesn't specify a "character set": it does precisely that in clause 5.1 by specifying "the basic character set" and by reference to 10646, "Universal Coded Character Set".

## Comment #5

Regarding encoding, it's appropriate to state in 5.1 that the encoding for *an SDL specification in a file* is not specified. But since an SDL description is specifying the format and interpretation of data bit streams, the encoding of strings is critical. Clause 6.6 addresses this.

However, I think the options in 6.6 for UCS encoding forms and encoding schemes could be a bit too limited. In particular, it does not permit use of the UTF-32 encoding form, and requires the byte order of UTF-16 encoding schemes to be disambiguated using BOM rather than being able to specify the encoding scheme directly in the SDL description. For instance, if one wanted to apply SDL to ISO/IEC 14496-22, then that would be a problem since that specifies that certain strings shall be represented in the UTF-16BE encoding scheme (cf. 11.3 of ISO/IEC 10646).

## Comment #6

Because this document is referring to UCS characters by numeric values (I'll assume code points are intended, not machine representations), 5.1 restricts far fewer characters than it probably should, and a lot of ambiguity is left.

1. 10646 specifies a number of non-character code points. They probably should not be permitted in an SDL description or in string values, but this document does not restrict these.

b) 10646 reserves code points D800 to DFFF for use in the UTF-16 encoding form. While it doesn't describe them as such, these are effectively non-character code points - that is, those code points will never be assigned to characters. Those \_*code points*\_ should not be referred to in an SDL description or in string literals, but this document does not restrict these.

c) Clause 5.1 cites U+0000 to U+001F as restricted "non-printable characters", but UCS contains many other non-printable characters.

* Other JTC1/SC2 standards specified "C0" and "C1" control characters, and 0000 to 001F correspond to the C0 control characters only. The C1 controls are encoded as 0080 to 009F; clause 5.1 excludes 0085, but all of 0080 to 009F should be excluded from SDL descriptions. Also, 007F is also classified in UCS as a control character and should be excluded from SDL descriptions.
* UCS also has 170 format control characters characters - see [this query](https://util.unicode.org/UnicodeJsps/list-unicodeset.jsp?a=%5B%3Agc%3DCf%3A%5D&g=&i=) for the complete set. Some of these can potentially be problematic in an SDL description.   
    
  In fact, some of these format controls can be \*\*dangerous\*\* in an SDL description if it is intended to be machine-readable; this is because the interpretation of the file by a human reviewer could be significantly different from the interpretation by a machine unless the tool used by the human reviewer implements particular security mitigations. See [Unicode Technical Standard #55, *Unicode Source Code Handling*](https://www.unicode.org/reports/tr55/) for details.
* UCS characters U+FFFC OBJECT REPLACEMENT CHARACTER and U+FFFD REPLACEMENT CHARACTER are not classified as control characters, but they should not be permitted in an SDL description: there's no useful scenario for them in an SDL description, and their use could only be confusing.
* UCS also includes other layout control characters that are not classified as control characters per se, but that are used to control visual presentation in certain situations and that, on their own, are not visible. At least the following probably should not be permitted in an SDL description:
  + U+00AD SOFT HYPHEN
  + U+034F COMBINING GRAPHEME JOINER
  + U+200B ZERO WIDTH SPACE
  + U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER
  + U+FE00..U+FE0F, U+E0100 VARIATION SELECTOR-1 .. U+E01EF VARIATION SELECTOR-256
  + Tag characters U+E0020 TAG SPACE .. U+E007F CANCEL TAG

[This query](https://util.unicode.org/UnicodeJsps/list-unicodeset.jsp?a=%5B%3ADefault_Ignorable_Code_Point%3A%5D&g=&i=) shows code points that are classified as Default Ignorable; any of the assigned characters are non-printing at least in isolation if not in all contexts.

## Comment #7

Clause 5.5 states that identifiers "may be comprised of ... Latin alphabetic characters". This clause doesn't mention that the set so-defined is not a stable set, but can grow and has grown over time with new UCS versions.

[This query](https://util.unicode.org/UnicodeJsps/list-unicodeset.jsp?a=%5B%3Asc%3DLatn%3A%5D%26%5B%3Agc%3DL%3A%5D&g=age&i=gc) lists all 1,448 Latin alphabetic characters grouped by age (Unicode version), showing how that set has grown over time. If an implementation is parsing an SDL description in a machine-readable file, then it could need to know whether it should expect a stable identifier space. [Unicode Standard Annes #31, *Unicode Identifiers and Syntax*](https://www.unicode.org/reports/tr31/), discusses such issues (summarized in the Introduction). If the intent that identifiers can only use the Latin letters specified as part of the "basic" character set defined in clause 5.1, then it should be explicit about that; but if identifiers can use other Latin characters in the UCS, then it should address the issue of stability.

That clause also states that identifiers may include "digits" but isn't explicit by what that means. Clause 5.1 lists "0 .. 9" as part of the "basic" character set; if that's all that's intended, then that should be stated explicitly. Otherwise, [this query](https://util.unicode.org/UnicodeJsps/list-unicodeset.jsp?a=%5B%3Agc%3DNd%3A%5D&g=&i=gc) shows all UCS characters classified as digits. (Btw, that set can also grow over time.)

# Topic #9: Improve the flexibility of class\_id

## Motivation

Currently class\_id is required to be a bit value and has a mandatory position in the bitstream. This is unusable for polymorphic class functionality in ISOBMFF.

Additionally the class\_id\_identifier is optional which means it may be invisible as a class member. This does not align well with the idea that after parsing the class structure is a mirror of the bitstream.

## Goal

Modify **class** syntax to support arbitrary encoding and location of *class\_id* and make the definition of *class\_id\_identifier* mandatory.

# Topic #10: Extra built-in operator

## Motivation

Currently it is not easy to parse arrays of polymorphic class structures using control flow techniques.

## Goal

Add new built-in operators (similar to lengthof()) which assist with control flow parsing: nextbits(), getbits() and isidof().

# Topic #11: Flexible class size encoding

## Motivation

Currently expandable classes support size encoding via sizeOfInstance. Problems with sizeOfInstance include:

* the name is sizeOfInstance but there are no “instances” in SDL
* the name is mandatory
* the encoding in the bitstream is mandatory
* the location in the bitstream is mandatory
* it does not support class sizes which are not integer number of bytes
* the size value does not include the *class\_id* nor the encoded sizeOfInstance

## Goal

Support arbitrary encoding and location of class size, support arbitrary name, support arbitrary class size boundary and support length in bits.

# Topic #12: Implicit end of parsing

## Motivation

SDL provides the concept of variable length arrays with the implicit keyword. The following problems are identified:

* The concept does not support variable length arrays of elementary types.
* The keyword “implicit” is somewhat confusing.
* The behaviour of implicit arrays is subtly different depending on whether they contain expandable classes or non-expandable classes and this is not discussed in the text.
* The behaviour of implicit arrays contained within a parent expandable class is not discussed i.e. in terms of “parse until end of class” semantics.
* The behaviour of implicit arrays with respect to “parse until end of stream” semantics is not discussed.
* Currently ISOBMFF misuses non-implicit arrays as variable length arrays by simply not defining a dimension.

## Goals

**Change 1**: Formally define the concept of “parse until end of stream”.

**Change 2**: Formally define the concept of “parse until end of class” for expandable classes i.e. where the length of the class is known.

**Change 3**: Using the defined “parse until” semantics to define parsing behaviour in relation to variable length arrays containing: elementary types, (non-)/polymorphic classes and (non-)/expandable classes (not supporting string).

Based on clarifying array parsing behaviour based on element type and “parse until” semantics, the concept of a specifically defined implicit array becomes redundant. Instead an array can be defined for classes and elementary with “parse until” semantics as follows:

* a[3] => fixed length
* a[0..3] => closed range
* a[3..] => partly open range
* a[] => open range

**Change 4**: Remove the keyword implicit and the associated concept and instead allow arrays to be defined with a fixed length or a closed, open or partly open range.

The following subtly different statements appear in the current text (bold added for clarity):

For an **(implicit)** array of such objects **(non-expandable classes)**, it is possible to implicitly determine the length by examining the validity of the *class\_id* of the class. Objects **(classes)** are inserted in the array as long as the *class\_id* can be properly resolved to one of the *class\_id* values defined in the base (if not abstract) or its derived classes. **(Note this is different behaviour for expandable class members.)**

Anywhere in the bitstream where a set **(implicit array)** 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 **(sizeOfInstance)** information. **(Note this is different behavior for non-expandable class members.)**

**Change 5**: Clarify and align these two described behaviours with the removal of the implicit array concept by instead indicating the behaviour is based on the array element type and not the array type.

Based on defining “parse until end of class” semantics it is possible to define the concept of optional members of a class. For example:

class A {

int(8) a;

optional int(8) b;

if (exists(b)) {

int(8) c;

optional int(8) d;

}

}

**Change 6**: Introduce the optional keyword to indicate trailing optional members of a class with differing behaviour based on the member type i.e. elementary type, class, polymorphic class, expandable class (not supporting string)

**Change 7**: Introduce the ability to test for the existence of a previous optional member of a class via a new build in operator exists().

# Topic #13: Entry point

## Motivation

Currently it is unclear that the top-level entry for SDL must be a class and cannot be an array due to global scope rules. Both ISOBMFF and the current example in the annex of the SDL standard incorrectly assume an array can be used.

## Goal

Clarify the top-level entry point in SDL must be a class and align this behaviour with ISOBMFF which currently has a special case for a box size of 0.

# Topic #14: Unexpected trailing bits

## Motivation

Nothing is stated in SDL currently about the parser behaviour for unexpected trailing bits. Based on the items relating to “parse until” semantics this scenario can be discussed more without needing to enforce a particular approach.

## Goal

Explicitly discuss the different scenarios in which a parser can encounter trailing bits and clarify which have defined behaviour (based on “parse until” semantics, fixed size classes and expandable classes) and which have undefined behaviour.

# Topic #15: Explicit parsing behaviour

## Motivation

Currently ISOBMFF structures are built on invalid and arbitrary usage and interpretation of SDL syntax and incompatible parsing behaviour is defined in text. Instead ISOBMFF should be able to easily and correctly use the correct SDL and parsing behaviour associated with that SDL syntax should appear in the SDL standard descriptive text.

## Goal

Ensure references in ISOBMFF regarding parsing behaviour can be replaced by SDL syntax and parsing behaviour described in SDL.

# Topic #16: Extended value constraints

## Motivation

The SDL does not provide mechanisms to define value constraints beyond expected values of elementary type members or array members extending a particular base class. This results in a large number of constraints being defined in textual descriptions rather than in a succinct syntax.

## Change #1

Define an SDL syntax to limit a member to be a subset of derived instances of a particular base class.

For example:

class A {

base\_class b; // b can only be derived\_class1 or derived\_class3

}

could be expressed as:

class A {

base\_class<{derived\_class1, derived\_class3}> b;

}

## Change #2

Define an SDL syntax to limit the derived class types and counts which may be members of an array.

For example:

class A {

base\_class b[3..]; // b must contain:

// 1 derived\_class1 member and

// 2 or more derived\_class3 members

}

could be expressed as:

class A {

base\_class b[3..,<derived\_class1[1], derived\_class3[2..]>];

}

## Change #3

The SDL also does not provide a mechanism to further constrain values of an existing SDL i.e. when one SDL specification makes use of another.

Define a new SDL constraints mechanism which is orthogonal to the existing syntax and which can be applied optionally and cumulatively and which can be used to extend the constraints of an existing SDL definition i.e. further constrain values, derived classes, array member types and array lengths.

# Topic #17: Usage Phases concept

## Motivation

SDL extract:

*abstract concept of a volatile storage area for variable values*

*Note 1 to entry: A computer program implementation would use computer memory for this and may choose to impose constraints such as storage size.*

For the first edition of SDL the definition of parsing was kept purposefully abstract. Further clarification is needed to:

* improve clarity of the intended parsing and evaluation behavior when using an SDL specification
* provide further clarity of initialized and uninitialized values of parsed and computed variables in the parsing context
* reduce the number of items described as “undefined behavior” improve clarity around “invalid SDL specifications” and error conditions

## Changes

*New Terms and Definitions*

It is proposed to add new terms and definitions as follows:

initialise

set the value of a variable in the parsing context for the first time

initialized

the value of a defined variable exists in the parsing context

uninitialized

the value of a defined variable does not exist in the parsing context

NOTE: this is only possible for computed variables as parsed variables are always defined with default value of 0 or “”

scanning phase

reading an SDL specification and validating it

processing phase

using a software implementation of a bitstream parser derived or generated from a valid SDL specification to parse a bitstream

scanning error

an error condition reported by an SDL scanner caused by an invalid SDL specification

parsing error

an error condition reported by an SDL processor when the bitstream does not conform to an SDL specification

processing error

an error condition reported by an SDL processor during parsing caused by flow control or expression evaluation e.g. accessing non-populated elements of a partial array, accessing uninitialized values

SDL scanner

a computer software implementation which can perform the scanning phase

SDL processor

a computer software implementation which can perform the processing phase

**NOTE:**

The term “scanning” of an SDL definition is proposed rather than “parsing” of an SDL definition in order that the word “parsing” can be used specifically for the behavior of a “parsing” a bitstream.

However, this might be problematic as we already have a software project called mpeg-sdl-parser. This should be renamed to mpeg-sdl-scanner if the above terminology is accepted.

*New Clause for Usage Phases*

It is proposed to add a new clause as follows:

**Usage Phases**

When using an SDL specification there are two phases: scanning and processing.

These phases are abstract concepts and are introduced to clarify the intended behavior of SDL. They should not be considered normative in this document and should not be considered mandatory in any usage of SDL or any SDL implementation.

The scanning phase consists of reading an SDL specification and validating it. This is expected to be performed by an SDL scanner implementation which would typically perform the following steps:

* Syntax Validation: tokenizing the SDL specification and constructed a lexical token tree whilst checking for syntax errors
* Semantic Validation: converting the lexical token tree into an abstract syntax tree whilst checking for semantic errors and undefined behavior scenario

The implementation details of an SDL scanner to perform these steps via tokenization, token tree creation, abstract syntax tree creation, syntactic analysis, semantic analysis, error reporting and input/output handling are not relevant to this document and are not defined.

Potential outcomes when scanning an SDL specification using an SDL scanner may be:

* no errors or warnings indicating the SDL specification is valid
* scanning errors indicating the SDL specification is syntactically or semantically invalid
* scanning errors indicating the SDL specification is syntactically and semantically valid but may result in undefined behavior scenarios or processing errors during the processing phase. These issues should be prevented by the SDL specification, the standard presenting the SDL specification, the SDL scanner implementation or an SDL processor implementation

When an SDL specification refers to global scope elements introduced in another SDL specification, all referenced elements should be collated before being used in the scanning phase. The collation and concatenation of multiple SDL specifications is not considered in this document.

The processing phase consists of using a bitstream parser derived or generated from a valid SDL specification to parse a bitstream. This is expected to be performed by an SDL processor implementation which would typically perform the following steps:

* Initialization: initializing the parsing context in accordance with the SDL specification, for instance defining any global constant computed variables and initializing their values in the parsing context
* Parsing: parsing the bitstream in accordance with the SDL specification, defining variables, evaluating flow control and expression statements in the SDL specification and adding or updating values in the parsing context as necessary. An SDL processor must be aware of a specific entry point for the SDL specification when commencing to parse the bitstream.

The implementation details of an SDL processor to perform these steps via code generation, compilation, execution, error handling, parsing context management, memory management, error reporting and input/output handling are not relevant to this document and are not defined.

Potential outcomes when processing using an SDL processor may be:

* no errors indicating processing was successful
* parsing errors indicating the bitstream is not conformant to the SDL specification
* processing errors or unexpected results indicating undefined behavior scenarios exist which are not prevented by the SDL specification, the standard presenting the SDL specification or the SDL processor implementation.

Note: The scanning and processing phases should not be conceptually equated with programming language compilation and execution phases. The SDL scanning phase does not output a binary executable, and the SDL processing phase does not use a binary executable output from the scanning phase. Validation checks during the SDL scanning phase are somewhat like static compile time checks. Additionally, validation checks during the SDL processing phase are somewhat like dynamic runtime checks. However, in both cases there are enough concrete differences that equating the concepts is not recommended.

*New Clause for Parsing Context*

It is proposed to add a new clause as follows:

**Parsing Context**

A parsing context is initialized during the initialization step of the processing phase and subsequently updated during the parsing step. It is an abstract concept and is discussed purely to clarify the intended behavior of SDL.

The implementation details of a parsing context are not relevant to this document and are not defined. Specifically, this document does not define:

* maximum limits in the parsing context for the items outlined in clause 5.20 SDL specification limits.
* constraints on the storage width of number values stored in the parsing context.
* encoding of code points for string values

During the initialization and parsing steps of the processing phase when parsed variable definitions are encountered they should be initialized with default values (0 for numeric values and “” for string values) or the literal value defined in the SDL specification.

During initialization and parsing when computed variables definitions are encountered they should either be left uninitialized or initialized with the literal value defined in the SDL specification.

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

*Modifications to Existing Text*

It is proposed to modify various statements throughout the existing text as follows:

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.

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

parsing context

abstract concept of a volatile storage area for variable values

Note 1 to entry: ~~A computer program~~ **An** implementation of an **SDL processor** would use computer memory for this and may choose to impose constraints such as total storage size**, number width limits and maximum values for the SDL specification limits outlined in clause 5.20 SDL specification limits.**

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

invalid SDL specification

SDL specification that is ~~syntactically valid but is nonetheless considered~~ invalid due to either **syntactic or** semantic **issues** rules or the fact that such an SDL specification would **cause errors during scanning or processing** be impossible to interpret or process

**Note 1 to entry: Certain scenarios in this document are explicitly identified as resulting in an invalid SDL specification.**

undefined behavior

behavior resulting in an undefined outcome **or processing error** based on the interpretation or processing of an SDL specification

Note 1 to entry: Certain scenarios in this document are explicitly identified as resulting in undefined behavior.

Note 2 to entry:  ~~A computer program implementation or a standard presenting an SDL specification~~ **An SDL processor implementation** may choose to define expected behavior in these scenarios.

unspecified constraint

constraint or limit on a SDL specification which would need to be defined for it to be viably **used** ~~processed in some manner by a computer program~~ **by an SDL scanner or processor implementation**

Note 1 to entry:  As the scope of the SDL does not cover **SDL scanner or processor implementation** ~~such processing~~, where appropriate such potential constraints are identified and explicitly left unspecified. **See** **clause 5.20**

Note 2 to entry: **An SDL scanner or processor implementation** ~~A computer program implementation~~ or a standard containing an SDL specification may choose to specify such constraints.

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

The array element access operator (i.e. ‘**[]**’) shall be used with positive 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 **or less than zero shall result in a processing error.** ~~is not defined by the SDL which can lead to undefined behavior.~~

The result of the modulus operator where the value of the second operand is zero **shall be a processing error** ~~is not defined by the SDL which can lead to undefined behavior.~~

~~The value of~~ division where the value of the second operand is zero **shall result in a processing error**. ~~is not defined by the SDL which can lead to undefined behavior.~~

~~A standard presenting an SDL specification which consists of global scope elements introduced in separate document sections or which refers to global scope elements in another SDL specification should define details on the interpretation of element ordering.~~ A standard presenting an SDL specification should also define the top level (or entry point) element that a parser **an SDL processor implementation** should expect when commencing to parse a bitstream.

**The order of global scope elements is irrelevant and references to global elements defined later in an SDL specification are allowed. Because of this SDL does not define the concept of forward declarations.**

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

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

**Accessing** non-populated elements of a partial array shall ~~not be accessed~~ **result in a processing error.**

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

NOTE  As some of those bit depths may be uncommon for floats, ~~the expected environment implementing a given SDL specification~~ **an SDL processor** must support the chosen bit depth.

**An SDL scanner or processor** ~~A computer program~~ implementation or a standard containing an SDL specification may choose to specify such details.

**An SDL scanner or processor** ~~A computer program~~ implementation or a standard containing an SDL specification may choose to ascribe such meanings to such character sequences.

The encoding of a utfstring value into a bitstream may be performed using either UTF-8 or UTF-16 and the encoding to use is not defined by the SDL which can lead to undefined behavior. **An SDL scanner or processor** ~~A computer program~~ implementation or a standard presenting an SDL specification may choose to define expected behavior in this scenario.

**An SDL scanner or processor implementation** ~~A computer program implementation of the SDL or a parser for an SDL defined bitstream~~ shall not perform any form of UCS character normalization, composition or decomposition.

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 **an SDL scanner or processor implementation or** the standard presenting the SDL specification.

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 multiple character numeric literal

·       Maximum number of concatenated adjacent multiple character numeric literals

·       Maximum character length of a string literal value

·       Maximum number of concatenated adjacent string literals values

·       Maximum number of dimensions in a multi-dimensional array

·       Maximum number of elements in each dimension of an array

**·       Storage width limits of number values stored in parsing context**

Elementary data type parsed variables, class variables, string parsed variables and array parsed variables may be defined within the body of a loop, however ~~it is considered~~ a **processing** error **will occur** if the parsing control flow results in the redefinition of such class scoped member elements.

For example, a ~~legacy device~~ **SDL processor implementation** can decode an expandable class up to the last parsed variable appearing in the version of the class declaration that the ~~device~~ **implementation** is aware of and can skip the unknown class data following the last known variable.

Accessing the value of a computed variable which has not yet been initialised with a value via the assignment operator (e.g. ‘**=**’) ~~is not defined by the SDL and can lead to undefined behavior.~~**shall result in a processing error.**

**A parsing error will occur if the length of a variable is not sufficient to store a default value. For example:**

**unsigned int(0) key = 64..128; // parsing error**

~~It is undefined behaviour if the length is zero and a range of values is defined.~~ **A processing error will occur if the length of a variable is specified using another variable and this length is not sufficient to store a default value.** For example:

unsigned int(3) keySize; // keysize is parsed as 0

unsigned int(keySize) key = 64..128; // ~~key is 0~~ **processing error**

The **if**-then-**else** clauses shall not define duplicate class member variables being declared simultaneously. For example, the following is an invalid SDL specification as it allows for the member variable foo to be declared more than once simultaneously:

EXAMPLE 8

class invalid\_class {

int bar = 2;

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**, a scanning error will occur**

}

**In some cases this will not be apparent until processing commences. For example:**

**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, a processing error will occur**

**}**

**}**

Additionally, **switch** clauses shall not define duplicate class member variables being declared simultaneously. For example the following ~~is invalid and~~ **may result in a processing error:**

EXAMPLE 12

unsigned int(32) code;

switch (code) {

case 0:

Foo f1;

// flow through to case 1

case 1: {

Foo f1; // invalid duplicate definition **if code == 0**

}

Elementary data type parsed variables, class variables, string parsed variables and array parsed variables may be defined within the body of a loop, however ~~it is considered an error condition~~ **a processing error will occur** if the ~~parsing~~ control flow results in the redefinition of such class scoped member elements. For example:

EXAMPLE 3

class Looped {

int(4) count;

computed int i;

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

int(8) offset; // if count > 1 a~~n~~ **processing** error ~~condition~~ will occur as

// offset is redefined

int(4) vertex[2]; // if count > 1 a~~n~~ **processing** error ~~condition~~ will occur as

// vertex is redefined

}

}

Looped looped;

~~Such a scenario is considered an invalid SDL specification~~

In the following example, as there is a conditional clause there is no possibility that the variables will be redefined and therefore no possibility this will result in a~~n error scenario~~ **processing error**:

EXAMPLE 8

class Looped {

int(4) count;

computed int i;

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

if (i == 0) {

int(8) offset0;

}

if (i == 1) {

int(8) offset1;

}

}

}

unsigned int(8) count; // the only value values for count here are 0 or 1

// anything else would/should cause ~~parsing to fail~~ **a processing error**...

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

int(8) offset1; // ... as this can only be declared 0 or 1 times

}

if (i == 0) {

int(8) offset2;

}

offset2 == 0

* Annex 1: Existing Text

As a summary of the current approach to the “parsing context”, presented below is a list of related statements extracted from the first edition (with currently accepted changes to the WD of the second edition).

**NOTE:** For those familiar with the concepts, this section can be skipped as it simply an audit of associated statements.

*Relating to parsing context*

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.

computed variable

variable whose value is only stored in the parsing context

declaration

sequential unit of 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

definition

sequential unit of tokens used to create a parsed or computed variable and its value, potentially causing data to be consumed from a bitstream or stored in the parsing context

literal value

value of a literal stored in the parsing context

value

data stored in the parsing context which may originate from a literal or an evaluated expression

parsed variable

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

parsing context

abstract concept of a volatile storage area for variable values

Note 1 to entry: A computer program implementation would use computer memory for this and may choose to impose constraints such as storage size.

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

Computed variables values (which are stored only in the parsing context) and parsed 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 undefined behavior.

As the SDL does not specify constraints on the storage width of number values stored in the parsing context it is expected that all number values will always be of sufficient width to accommodate common value coercion and there is no need to define any behaviour when converting between signed and unsigned types.

As the SDL does not specify constraints on the storage width of number values stored in the parsing context but it does specify the width of parsed variables, an SDL specification shall not use a literal value to initialise an elementary type parsed variable that is outside of the number limit range of a parsed variable type.

The exact ordered set of code points stored in the bitstream shall be returned by a parser, even if the encoding of the data is changed when stored in the parsing context e.g., UTF-8 to UTF-16, UTF-16 Big Endian to UTF-16 Little Endian.

*Relating to unspecified/undefined behavior*

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 SDL specification or will result in undefined behavior.

invalid SDL specification

SDL specification that is syntactically valid but is nonetheless considered invalid due to either semantic rules or the fact that such an SDL specification would be impossible to interpret or process

undefined behavior

behavior resulting in an undefined outcome based on the interpretation or processing of an SDL specification

Note 1 to entry: Certain scenarios in this document are explicitly identified as resulting in undefined behavior.

Note 2 to entry:  A computer program implementation or a standard presenting an SDL specification may choose to define expected behavior in these scenarios.

unspecified constraint

constraint or limit on a SDL specification which would need to be defined for it to be viably processed in some manner by a computer program

Note 1 to entry:  As the scope of the SDL does not cover such processing, where appropriate such potential constraints are identified and explicitly left unspecified.

Note 2 to entry: A computer program implementation or a standard containing an SDL specification may choose to specify such constraints.

The array element access operator (i.e. ‘**[]**’) shall be used with positive 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 can lead to undefined behavior.

The modulus operator (i.e. ‘**%**’) shall not 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 can 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 can lead to undefined behavior.

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

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 can lead to undefined behavior.

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 multiple character numeric literal

·       Maximum number of concatenated adjacent multiple character numeric literals

·       Maximum character length of a string literal value

·       Maximum number of concatenated adjacent string literals values

·       Maximum number of dimensions in a multi-dimensional array

·       Maximum number of elements in each dimension of an array

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

For example, the standard presenting the SDL specification can state that a class declared using an class\_id\_range or extended\_class\_id\_range is effectively abstract, i.e. class\_id values encountered in the bitstream are values appearing as single class\_id values in derived class declarations. Alternatively, the standard can 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 results 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.

Elementary data type parsed variables, class variables, string parsed variables and array parsed variables may be defined within the body of a loop, however it is considered an error condition if the parsing control flow results in the redefinition of such class scoped member elements.

Such a scenario is considered an invalid SDL specification. If such loop-based variable population is required, it is preferred to use partial arrays.

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

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

It is undefined behaviour if the length is zero and a range of values is defined. For example:

unsigned int(3) keySize; // keysize is parsed as 0

unsigned int(keySize) key = 64..128; // key is 0

*Relating to uninitialised/initialised values*

Parsed elementary data type variables with definitions that fall outside the flow of parsing shall have an initial value of 0.

Parsed string variables with definitions that fall outside the flow of parsing shall have an initial value of an empty string (e.g. ‘""’).

For this purpose, an extended array definition shall be used in which individual elements of the array may be accessed for population i.e. allowing the definition of dynamically sized sparse arrays. Non-populated elements of a partial array shall not be accessed.

*Relating to scope*

The scope of an identifier is the part of an SDL specification within which the identifier is visible and may be referenced. An identifier shall 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. The scope of a global identifier commences at the point it is declared or defined and terminates at the end of the SDL specification. The only items that may be present in global scope are:

·       constant computed elementary variable definitions

·       map declarations

·       class declarations

Note that all class declarations and map declarations can only be made at global scope.

A standard presenting an SDL specification which consists of global scope elements introduced in separate document sections or which refers to global scope elements in another SDL specification should define details on the interpretation of element ordering. A standard presenting an SDL specification should also define the top level (or entry point) element that a parser should expect when commencing to parse a bitstream.

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 also defines two specific scope behaviors for parsed and computed variables which differ from other well-known programming languages. These are presented in the following two clauses.

Parsed variables defined within a class declaration, regardless of nested block scopes or conditional branch scopes have class scope, i.e., they are available as class member variables.

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

Parsed variables defined within a class declaration, regardless of conditional branch scopes have class scope, i.e., they are available as class member variables.

Unlike other languages variables can be defined within switch clauses without defining a new scope. However computed variables may not be defined more than once without introducing a new scope.

Computed variables may be defined within the body of a loop as they are block scoped.

Elementary data type parsed variables, class variables, string parsed variables and array parsed variables may be defined within the body of a loop, however it is considered an error condition if the parsing control flow results in the redefinition of such class scoped member elements.

# Topic #18: const for parsed variables

## Background

The semantics of the const keyword for **computed variables** in SDL has always been clear:

* Computed variables may not be redefined (within the same scope).
* Mutation of computed variables is allowed by default.
* The const keyword disallows mutation of computed variables.

These semantics align directly with the usage of the const keyword in other programming languages.

In contrast, the semantics of the const keyword for **parsed variables** in SDL has never been clear due to the lack of detail in the brief text appearing within 14496-12.

It has been discovered that the original intent of the semantics of the const keyword for parsed variables in SDL is more clearly explained in documents available at <https://flavor.sourceforge.net>. Unfortunately, these semantic explanations were discovered after publication of the first edition of 14496-34 and the intended semantics differ.

The semantics of the const keyword for **parsed variables** in SDL as per 14496-34 are:

* Parsed variables may not be redefined within class scope (as parsed variables have class instead of block level scope they must not be re-defined within inner control flow loops but can be redefined in alternative control flow branches).
* Mutation of parsed variables is allowed by default.
* The const keyword disallows mutation of parsed variables.

Although the first point above is somewhat unique to SDL (and is discussed further below), the latter point is aligned directly with the usage of the const keyword for computed variables and also generally in other programming languages.

The original intent of the semantics of the const keyword for **parsed variables** in SDL appear to be as follows:

* Parsed variables may be redefined within class scope (as parsed variables have class instead of block level scope they can be re-defined within inner control flow loops and can be redefined in alternative control flow branches).
* Mutation of parsed variables is disallowed always.
* The const keyword disallows re-definition of parsed variables.

The latter point is completely unaligned with the usage of the const keyword for computed variables and in other programming languages. This is where lack of clarity has stemmed from.

## Motivation

In hindsight it makes SDL much simpler to always disallow mutation of parsable variables. The main reason for defining them as mutable in 14496-34 was to attempt to incorporate the const keyword as it was known to exist in the syntax; but to incorporate it with the semantics already associated with computed variables and other programming languages.

It is proposed that a better approach is to modify the SDL in the next edition of 14496-34 so that:

* parsable variables are always immutable.
* the const keyword is no longer supported for parsable variables (but is still supported for computed variables).

Immutable parsable variables will have a significant simplifying effect in comprehending the standard and making practical use of it. For example:

* The de-facto understanding in the MPEG community is that parsed variables are not mutated and always retain the same storage width as when parsed.
* Allowing parsed variable mutation can lead to problems with the ability to reproduce the same bitstream after parsing due to lost values or different value widths.
* Allowing parsed variable mutation does not align well with the concept of the parsed result being a “full picture” of the parsed bitstream.

The changes to the current working draft can be provided at a subsequent meeting if the above is accepted.

## Changes

Because SDL currently allows parsed variable mutation it was necessary to define “deep copy” semantics when using parsed variables as parameters to classes. From the text:

Values are copied (including a deep copy of all values for nested class structures) when provided as parameter values to a class meaning that any changes to parameter values by the class making use of them are not visible externally. As an example:

EXAMPLE 12

class A {

int(2) i;

}

class B (A a) {

int(2) j;

a.i++;

int k = a.i;

}

A a;

// assume a.i == 2 here

B b(a);

// a.i == 2 here as changes within b to a.i were on a deep copy of a

// b.k == 3 here as b.k was initialised using the modified deep copy of a

If only computed variables were to be mutable it is proposed to state that parameter values are passed by reference – which should be clearer and simpler.

## Redefinition of Parsed Variables

If the above points are accepted, the remaining semantic point for discussion surrounds the ability to re-define parsable variables.

To serve as a reminder to the currently defined semantics in 14496-34 some extracts are presented below:

A parsed 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 parsed 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;

The **if**-then-**else** clauses shall not define duplicate class member variables being declared simultaneously. For example, the following is an invalid SDL 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

}

}

Elementary data type parsed variables, class variables, string parsed variables and array parsed variables may be defined within the body of a loop, however it is considered an error condition if the parsing control flow results in the redefinition of such class scoped member elements. For example:

EXAMPLE 3

class Looped {

int(4) count;

computed int i;

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

int(8) offset; // if count > 1 an error condition will occur as

// offset is redefined

int(4) vertex[2]; // if count > 1 an error condition will occur as

// vertex is redefined

}

}

Looped looped;

If such loop-based variable population is required, it is preferred to use partial arrays. Partial array parsed variables may be defined within the body of a loop as effectively these are accessing specific elements within a single defined array. For example:

EXAMPLE 4

class Looped {

int(4) count;

computed int i;

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

int(8) offsets[[i]];

int(4) vertices[[i]][2];

}

}

Looped looped;

// here, looped.offsets is an array with a size of looped.count

// here, looped.vertices is a 2-dimensional array with sizes of looped.count,2

The main goal with these semantics in 14496-34 is to ensure a parsed value is always accessible. If parsed variables were to be re-defined, the previous parsed value would no longer be accessible or known. This does not align well with the concept of a parser having a “full picture” of the parsed bitstream.

NOTE: As flow control may rely on parsed values, such re-definition scenarios may not be discernible until the bitstream is being parsed. Such scenarios are covered in the input document: **m73968 [SDL] Addition of Usage Phases concept to ISO-IEC 14496-34**

As mentioned earlier, the original intent of SDL was to allow re-definition of parsed variables (i.e. allow previously parsed values to no longer be accessible) and to prevent this on an ad-hoc basis via the usage of the const keyword. It is proposed that the const keyword should not be used for this purpose as the semantics clearly differ from those for computed variables and in other programming languages.

It is proposed to continue to disallow the re-definition of parsed variables as currently defined in 14496-34.

NOTE: There is no currently known scenario where disallowing the re-definition of parsed variables will make SDL unusable. It is anticipated that if the efforts to align the formalised 14496-34 version of SDL with the 14496-12 usage for Boxes succeed, this will prove the fact that re-definition of parsed variables is not required.