# INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
# ORGANISATION INTERNATIONALE DE NORMALISATION
# ISO/IEC JTC1/SC29/WG11
# CODING OF MOVING PICTURES AND ASSOCIATED AUDIO

**ISO/IEC JTC1/SC29/WG11**
**MPEG96/M0856**
**March 22, 1996**
**Florence, Italy**

**Source:** **Alexandros Eleftheriadis and Yihan Fang, Columbia University***
**Title:** **A Revision and Proposed Grammar for the MSDL Syntactic Description Language**
**Purpose:** **Proposal**
**Subgroup:** **MPEG-4 Integration/MSDL**

## 1. Introduction

This document addresses some of the ambiguities and shortcoming of MSDL Syntactic Description Language (MSDL-S—the bitstream syntax specification part of MSDL), as described in document MPEG96/N1164 [1], and proposes some modifications that streamline its definition. In addition, it introduces a formal grammar for the definition of the language as a basis for unambiguous definition of the syntactic constructs it allows.

## 2. Scope of MSDL-S

MSDL-S addresses the need to disengage the definition of the bitstream syntax of MPEG-4 content from the decoding and/or object rendering tools. This requirement originates from the fact that a given syntax specification may be decoded using different implementations of the relevant algorithms. This certainly has been the central theme in the MPEG-1 and MPEG-2 series of specifications, eventhough the syntax specification utilized both formal and non-formal techniques (i.e., it included explanatory text without which the definition of the syntax would be incomplete).

The flexibility and programmability aspects of MPEG-4 provide content developers the opportunity to create customized bitstream structures to suit their specific application needs. In order to promote an open approach in terms of bitstream definition, in which content developers may wish to publish their low-level syntax but not their proprietary processing algorithms, it is then important for MPEG-4 to have a facility that will make this separation of syntax and processing possible as well as easy.

An additional important benefit of separating bitstream parsing from processing is that the task of the bistream architect is greatly simplified. Focus is drawn on the important tasks of decoding information and preparing it for display, and not in the mundane task of obtaining bits from a

* Email: {eleft,fang}@ee.columbia.edu, Address: 1312 Mudd Bldg., Department of Electrical Engineering, Columbia University, New York, NY 10027, USA. This document represents work in progress.

bitstream. This is an underlying theme in all typed programming languages, where a set of standard types (chars, ints, doubles) are directly provided by the language, without requiring direct manipulation of their representation by the programmer. The language compiler or interpreter is responsible for ensuring that data manipulations (including conversions) are consistent with type declarations, or their derivatives (extended types). These languages, of course, do not take into account that the data may be obtained from the bitstream, but the same concept applies.

Finally, separation of the bitstream syntax from decoding provides for automatic compliance with the overall bitstream architecture that MSDL will define (multiplexing level). The alternative approach would put the burden of complying with bitstream-level object delineation and naming to the content developer, whereas MSDL-S can provide automated facilities that prohibit mistakes or want about potential conflicts.

We should note that Level 1 MSDL is not possible if the syntax is intertwined with processing. This is because the way to connect existing tools and algorithms is by using new data types or objects; if parsing of these objects requires processing, this implies that a virtual machine must be defined, which in turn makes Level 1 equivalent to Level 2.

## 3. The relationship between MSDL-S and MSDL-R

MSDL-R [1] addresses the general programming facilities of MSDL, and is the language in which decoding and generic processing operations are described. MSDL-S is a an orthogonal subset of MSDL-R. Orthogonality here implies that the two are independent: the specification of MSDL-S does not affect MSDL-R and vice versa. There are, of course, some commonalities assumed, at the level of their capability to define object hierarchies.

In its current form (originating from [2]), MSDL-S (in its textual form) assumes a C++ or Java-like approach as the theme of MSDL-R. It then proceeds to extend the typing system by providing facilities for defining bitstream-level quantities, and how they should be parsed. MSDL-S can be seen as generalizing the concept of declaring constants with hard-coded values, to that of declaring constants that obtain their values from a bitstream. Similarly to traditional constants where a programmer is not concerned how the initialization is performed, but needs to assume that it is performed before the variable is accessed, an MSDL-R programmer assumes that a constant is parsed from the bitstream before it is accessed.

The interface between MSDL-R and MSDL-S is therefore very well-defined. In addition, it poses no restrictions to the structure of either MSDL-R or MDSL-S. The rule of guaranteeing that a variable is parsed before it is accessed is the most general one. Simpler rules can also be adopted; an example would be to mandate that a variable is parsed when the object it is part of is instantiated. This would be an implementation detail, however, and the more general rule can be used without limiting the specification in any way.

In the following we describe the proposed revision of MSDL-S in detail. A formal grammar is included at the end of this document (Appendix A).

## 4. Syntactic Description Language

This section describes the syntactic construction features of MSDL, in the form of formal rules. It directly extends the C-like syntax used in MPEG-1 and MPEG-2 into a well-defined framework that lends itself to object-oriented data representations. We first describe elementary constructs, moving to composite syntactic constructs, arithmetic and logical expressions, and finally address syntactic control flow. The latter is needed to take into account context-sensitive data. Several examples are used to clarify the structure using the MPEG-2 Video IS.

### 4.1 Elementary Data Types

We can in general identify the following syntactic elements:

1. Constant-length direct representation bit fields or Fixed Length Codes (e.g., temporal_reference). These include the encoded value as it is to be used by the decoder.
2. Variable length direct representation bit fields, sometimes called variable length FLCs. These are FLC for which the length is determined by the context of the bitstream (e.g., another parameter).
3. Constant-length indirect representation bit fields (e.g. chroma_format or coded_block_pattern). These require an extra lookup into an appropriate table or variable, or some algorithmic processing to obtain the desired value (e.g., coded_block_pattern).
4. Variable-length indirect representation bit fields (e.g., Huffman codes for DCT coefficient run-lengths).

#### *4.1.1 Constant-Length Direct Representation Bit Fields*

These can be simply represented as:

| |
|---|
| Rule 1<br>[aligned] *type*[(*length*)] *element_name* [= *value*];      // C++-style comments allowed |

The *value* attribute is only present 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* are always present, except if the data is non-parsable, i..e., it is not included in the bitstream. The attribute 'aligned' means that the data is aligned on a byte boundary. As an example, a start code would be represented as:

```
aligned const(32) picture_start_code=0x00000100;
```

An entity such as temporal reference would be represented as:

```
unsigned int(5) temporal_reference;
```

where 'unsigned int(5)' indicates that the element should be interpreted as a 5-bit unsigned integer (by default with the most significant bit first—MSBF). Two non-standard data types are 'bit' (just a sequence of bits) and 'vlc', which we discuss below.

*4.1.2 Variable Length Direct Representation Bit Fields*

This case is covered by Rule 1, by allowing the *'length'* field to be a variable included in the bitstream,or an expression involving such a variable. For example:

```
unsigned int(3) precision;
int(precision) DC;
```

*4.1.3 Constant-Length Indirect Representation Bit Fields*

Here, in addition to the actual element we need to define how it is mapped to obtain the actual values that the decoder will use. This can be accomplished by defining the map itself:

Rule 2
map *MapName* (type_idx, *type1 name1*, *type2 name2*, …, *typeN nameN*) [sign=*col*] {
*index*[s] *value_1 … value_M,*
[*index*[s] *value_1 … value_M, …*]
};

Note that this is a multi-dimensional mapping: there is one input (index), and zero or more outputs (values), allowing the table to be used to specify the value for several different variables (VLCs typically encode several events jointly for efficiency reasons). The *type* entries in the definition indicate the type of the value columns, which are addressed using the *'name'* information (e.g., *MapName.name1)*. The *index* entry is always specified as a series of bits. If there are no value columns, then the default values are simply the position of the *index* in the table (starting from 0).

As an example, we have:

```
// a table that relates the chroma format with the number of blocks per signal component
map blocks_per_component (uint(2), uint Yblocks, uint Ublocks, uint Vblocks) {
        0       4       1       1,      // 4:2:0
        1       4       2       2,      // 4:2:2
        2       4       4       4       // 4:4:4
};
map(blocks_per_component) chroma_format;  // this is a fictional example
```

Using the above declaration, we can access a particular value of the map using: blocks_per_component.Ublocks[chroma_format]. In other words, chroma_format can be used as a regular index into the columns of the map. An alternative is: chroma_format.Ublocks.

*4.1.4 Variable Length Indirect Representation Bit Fields*

For a variable length element utilizing a Huffman table, a similar specification is used:

map(*table*) ac_dct_coefficient;

The definition of the table is done in exactly the same way as we saw before, but here we have the data type ‘vlc’ as the index data type. For example:

```
map sample_vlc_map (vlc, int(1) value1, int value2)  {
        0b0000.001     0        5,
        0b0000.0001    1        -14
};
```

The vlc codewords are binary strings (starting with ‘0b’), optionally using the period (‘.’) every four digits for readability.

Very often, VLC tables are incomplete: due to the large number of possible entries, it is 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 representation. To allow for such exceptions, element specifications are allowed for map values. This is illustrated in the following example:

```
map sample_map_with_esc (vlc, uint value1, uint value2)  {
        0000.001        0             5,
        0000.0001       1             -14,
        0000.0000.1     map(foo)      int(32),
        0000.0000.0     0             -20
};
```

As written above, when the codeword 0b0000.0000.1 is detected in the bitstream, then another VLC table is used for the value for column 1, while the following 32 bits will correspond to the value for column 2 (the order is significicant). Using this construct, the complete behavior of the VLC mapping is described in a concise manner in a single place.

A notational convenience that is present in VLC tables is use of the same codewords for both the negative and positive value of the variable represented. This can be accommodated by denoting in the codewords the position of the sign bit, using the letter ‘s’. Typically, the sign change affects only one column (at least this is the case with existing MPEG VLC tables). Hence we only need to denote the column that is affected by the sign bit. This is done as shown below:

```
map table_with_signs (vlc, unsigned int, int) sign=2 {
        000            1       5,
        0000.s         2       7,
        0001.00s       3,      -10,
        0001.010       4,      4
};
```

The above specification says that the sign bit (1 for positive, 0 for negative) affects column 2 of the values. Hence 0000.1 results in a value of 7, while 0001.000 results in a value of 10 (-(-10)).

### 4.2 Composite Data Types

Equipped with the above definitions for fundamental types, we can now examine the definition of composite types or objects. A very useful feature is to be able to immediately identify the type of object that we are dealing with; object identifiers are then a particularly attractive feature. In several cases, the desire for bit efficiency precludes their use (this is the case in MPEG-2 below the slice level). The definition of a composite object can then be expressed as:

Rule 3

```
    class object_name [is parent_class] [:
                [aligned] bit(length) [id_name]= object_id | id_range ] {
        [element; …]            // zero or more elements
};
```

The different elements are definitions of elementary bitstream components as we saw in section 4.1, or control flow that is discussed in the following section. The *object_id* is optional, and if present is the key multiplexing entity for individual objects. The *id_range* is specified as *start_id* .. *end_id*, inclusive of both bounds, to express that the object can have a range of possible id's. The optional 'is *parent_class*' specifies that the class can be used wherever a slot of type *parent_class* has been specified. Using *parent_class* defines an implicit switch on class id's in the decoding process. This switch is totally implicit, as it is not needed to define the possible structure of the bitstream, only to decode it.

Examples:

```
    class foo {
         ...
        bar name_1;
        ...
    }

    class bletch is bar { ... } // this class can be used in the name_1 slot

    class slice: aligned uint(32) slice_start_code=0x00000101 .. 0x000001AF {
        … // here we get vertical_size_extension, if present
        if (scalable_mode==DATA_PARTITIONING) {
            unsigned int(7) priority_breakpoint;
        }
        …
    } ;
```

The order of declaration of bitstream components is important: it is the same order in which the elements appear in the bitstream.

We can also encapsulate objects within other objects. In this case, the *element* mentioned at the beginning of this section is an object itself.

### 4.3 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.

### 4.4 Arithmetic and Logical Expressions

All standard arithmetic and logical operators of C/C++ are used as is, including their precendence rules.

### 4.5 Temporary Variables

In order to accommodate complex syntactic constructs, in which context information cannot be directly obtained from the bitstream but is the result of a non-trivial computation, a small number of temporary variables are provided. These are named $a through $z, and can hold any variable type up to a maximum length of 128 bits. They can 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];
…
for ($i=0, $n=0; $i<size; $i++) {
        if (array[$i]!=0)
                $n++;
}
int(3) coefficients[$n];// read as many coefficients as there are zero elements in
                        // array[]
```

### 4.6 Syntactic Control Flow

The syntactic control flow 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.

```
Rule 4
        if (condition) {
                …
        } [ else if  (condition) {
        …
        }] [else {
                …
        }]
```

The following example illustrates the procedure.

```
class conditional_object {
        unsigned int(3) foo;
        int(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’. Another example is:

```
class conditional_object {
        unsigned int(3) foo;
        boolean bar_flag;
        if (bar_flag) {
                unsigned int(8) bar;
        } else {
                map(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.

In the same category of context-sensitive objects we have the so-called repetitive 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. To eliminate the need for temporary variables (counters etc.), we use the following structures

Rule 5

The construct '[*bitstring*]' is a test condition that is true (non-zero) if the next bits present in the input bitstream are equal to *bitstring*. The contruct '[*bitstring**]' performs the same operation, but if the string is found, the bits are removed from the bitstream.

Rule 6

for (*expression1*; *expression2*; *expression3*) {
… // identical to a C/C++ for(;;) loop
}
or
do {
… // repeat at least once, and while *condition* is true
} while (*condition*);
or
while (*condition*) {
… // repeat zero or more times, and while *condition* is true
}

In Rule 6 we have three different forms: a fixed repetition using the *times* argument, or a variable number of repetitions dependent on the value of a *condition.* A condition is considered to evaluate to true if it reduces to a non-zero value. The C/C++ logical operators can be used to construct complex conditions.

**4.7 Parsing Modes**

Typically, parsing proceeds from the top-most class which is identified with the prefix 'main'. In several cases, the precise order of objects is not specified by the syntax. This is especially true for high-level structures, but can also occur in low-level components (e.g., in MPEG-2 NBC Audio). This implies that any one from a set of object types can be present in the bitstream, without violating the syntax specification. To accommodate this behavior, the following construct is introduced:

[ascending|descending] list {[*class1, class2, …*]} *name*;

The elements inside the braces (*class1* etc.) have to be classes with definitions that include object id's. The syntax here indicates that the bitstream contains any number of the objects contained inside the braces, in any order. If no such object is found, then control breaks out of the 'list {}' loop. If the list is empty, then the entire list of objects currentlty known to the MPEG-4 receiver is acceptable.

If the objects specified in the list have id ranges, it makes sense to specify monotonic increase (*ascending*) or decrease (*descending*) of those id's. This means the list includes the sequence of objects that have monotonically changing id's and the first violation of ordering terminates the list right before the violating element.

### 4.8 References and Scoping Rules

*Inheritance*
When a class is defined as a subclass of another, declarations made in the superclass are visible in the subclass.

*Temporary Variables*
The scope of a temporary variable can only be within the class that it is used. It can not be inherited to its child classes.

*Data Member Scoping Rules*
When a data member is defined in a class, the following scoping rules are used:
1. The data member can be a constant number or array or map.
2. The data member can be another data element, which means this is a context-dependent reference. In this case it can reference either a data member of all superclasses, or a public static or constant data member of another class.
3. The data member can be an expression of local temporary variable(s) and the above 2 cases.

*Variable Scoping Rules*
When a variable is referenced inside a method definition, the following scoping rules are used:
1. The current block is searched first, and then all enclosing blocks, up to and including the current method. This is considered the local scope. After the local scope, the search continues in the class scope.
2. The variables of the current class are searched.
3. If the variable is not found, variables of all superclasses are searched, starting with the immediate superclass, and continuing up through class objects until the variable is found. If the variable is not found, a compile-time error is generated.

## References

[1] ISO/IEC SC29/WG11 N1164, MSDL Specification, Version 1.0, January 1996, Munich, Germany.
[2] A. Eleftheriadis, Columbia University, ISO/IEC SC29/WG11 Contribution M0546, “A Syntactic Description Language for MPEG-4”, November 1995, Dallas, TX.

**Appendix: A Formal Grammar for the MSDL Syntactic Description Language**

```
//*****************************************
//   Meta-notation :
//
//   | for alternation,
//   (...) for grouping,
//   postfix ? for 0 or 1 occurrences,
//   postfix + for 1 or more occurrence,
//   postfix * for 0 or more occurrences.
//*****************************************
```

CompilationUnit:
   Typedeclaration*
;

TypeDeclaraton:
   ClassDeclaration
;

ClassDeclaration:
   Modifier* **'class'** ClassName (**'is'** ClassName)?
   ( **':'** (**'aligned'**)? **'bit'** **'('** BitFieldLength **')'**
    (`Identifier`)? **'='** ( ObjectID | ObjectIDRange ) )?
   **'{'** FieldDeclaration* **'}'**
;

FieldDeclaration:
   MethodDeclaration
| ConstructorDeclaration
| VariableDeclaration
| StaticInitializer
| ';'
;

MethodDeclaration:
   Modifier* Type `Identifier` **'('** ParameterList? **')'**
   ( **'{'** Statement* **'}'** | **';'** )
;

ConstructorDeclaration:
   Modifier* `Identifier` **'('** ParameterList? **')'**
   **'{'** Statement* **'}'**
;

VariableDeclaration:
   Modifier* Type ( **'('** BitFieldLength **')'** )?
   VariableDeclarator (**','** VariableDeclarator)* **';'**

```
;

VariableDeclarator:
   Identifier ArrayDeclarator?    ('=' VariableInitializer)?
;

VariableInitializer:
   Expression
|  '{' ( VariableInitializer ( ',' VariableInitializer )* ','? )? '}'
;

ArrayDeclarator:
   '[' ']'
|  '[' Expression ']'
|   ArrayDeclarator '[' Expression ']'
;

StaticInitializer:
  'static' '{' Statement* '}'
;

ParameterList:
   Parameter (',' Parameter)*
;

Parameter:
   TypeSpecifier Identifier
;

Statement:
   VariableDeclaration
|  Expression ';'
|  '{' Statement* '}'
|  'if' '(' Expression ')' Statement ('else' Statement)?
|  'for' '(' Expression ';' Expression ';' Expression ')' Statement
|  'while' '(' Expression ')' Statement
|  'do' Statement `while' '(' Expression ')' ';'
|  'return' Expression? ';'
|  'list' '{' ClassName ( ',' ClassName)* Identifier ';' '}'
|  'map' Identifier  '(' IndexType (',' Type Identifier )* ')'
   '{' ObjectID   (Expression)* ( ',' ObjectID (Expression)*)* '}' ';'
;

IndexType:
   Type ( '(' BitFieldLength ')' )?
;
BitFieldLength:
```

```
    Expression
;

Expression:
   Expression '+' Expression
|  Expression '-' Expression
|  Expression '*' Expression
|  Expression '/' Expression
|  Expression '%' Expression
|  Expression '^' Expression
|  Expression '&' Expression
|  Expression '|' Expression
|  Expression '&&' Expression
|  Expression '||' Expression
|  Expression '<<' Expression
|  Expression '>>' Expression
|  Expression '=' Expression
|  Expression '+=' Expression
|  Expression '-=' Expression
|  Expression '*=' Expression
|  Expression '/=' Expression
|  Expression '%=' Expression
|  Expression '^=' Expression
|  Expression '&=' Expression
|  Expression '|=' Expression
|  Expression '<<=' Expression
|  Expression '>>=' Expression
|  Expression '<' Expression
|  Expression '>' Expression
|  Expression '<=' Expression
|  Expression '>=' Expression
|  Expression '==’ Expression
|  Expression '!=' Expression
|  Expression ',' Expression
|  Expression '.' Expression
|  Expression '?' Expression ':' Expression
|  Expression '[' Expression ']'
|  Expression '++' Expression
|  Expression '--' Expression
|  Expression '++'
|  Expression '--'
|  '-' Expression
|  '!' Expression
|  '~' Expression
|  '(' Expression ')'
|  ‘[‘ Expression ‘]’
|  '(' Type ')' Expression
```

```
| Expression '(' ArgList? ')'
| Identifier
| Number
| String
| Character
;

ArgList:
  Expression (',' Expression )*
;

Type:
  TypeSpecifier
;

TypeSpecifier:
  'boolean'
| 'bit'
| 'char'
| 'uint'
| 'int'
| 'short'
| 'long'
| 'float'
| 'double'
| ‘vlc’
| ‘map’
| ClassName
;

Modifier:
  'public'
| 'private'
| 'protected'
| 'static'
| 'const'
| ‘aligned’
;

ClassName:
  Identifier
| ClassName ‘.’ Identifier
;


ObjectIDRange:
  ObjectID ‘..’ ObjectID
```

;

ObjectID:
  Expression
;

```
Identifier:
     [a-zA-Z_$][a-zA-Z0-9_]*

Number:
     -?([0-9]+)|([0-9]*\.[0-9]+([eE][+-]?[0-9]+)?)

Character:
     .
String:
     .*
```