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/M1262
Chicago, USA

Title: **A Proposed Revision of MSDL-S**
Source: **Yihan Fang and Alexandros Eleftheriadis (Columbia University)**
Status: **Proposal**
Group: **MPEG-4 Integration/MSDL**
Date: **September 26, 1996**

1.1 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 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.

MSDL-S has been designed so that it can describe existing standards. It is also an easy-to-read way of defining syntax specifications, since it is based on a set of well-defined elements with unambiguous semantics. In a large extend, it is purely declarative; some procedural facilities are provided, however, to cover primarily complicated cases of entropy coding.

1.2 The relationship between MSDL-S and MSDL-R

MSDL-R 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 commonalties assumed, at the level of their capability to define object hierarchies.

In its current form, MSDL-S (in its textual form) assumes a C++ or Java-like approach as the central 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 could also be adopted; an example would be to mandate that a variable is parsed when the object 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 in Section 5.4.

1.3 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 and functions. Syntactic flow control is needed to take into account context-sensitive data. Several examples are used to clarify the structure, primarily based on the MPEG-2 Video IS.

1.3.1 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).
5. Arithmetically coded values.

1.3.1.1 Constant-Length Direct Representation Bit Fields

These can be simply represented as:

Rule 1

[aligned] *type*[(*length*)] *element_name* [= *value*];

// C++-style comments allowed

The *type* is one of the following: 'int' for signed integer, 'unsigned int' or 'unsigned int' for unsigned integer, 'bit' for raw data. '*length*' indicates the length of the element in bits, as it is stored in the bitstream. 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 bit(32) picture_start_code=0x00000100;
```

An optional numeric modifier, e.g. aligned(32), can be used to signify alignment on other than byte boundary.
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).

Note that constants are defined using the 'const' attribute:

```
const int SOME_VALUE=255;
const bit(3) BIT_PATTERN=1;
           // this is equivalent to the bitstring "001"
```

To designate binary values, the '0b' prefix is used, similar to the '0x' prefix for hexadecimal numbers, and a period ('.') can be optionally placed every four digits for readability. Hence 0x0F is equivalent to 0b0000.11111.

1.3.1.2 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;
```

1.3.1.3 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* (*output_type*) {
    index, *value, ...*
};

The index type of such a table is always 'bit'. These tables are used to translate or map bits from the bitstream into a set of one or more values. The *output_type* entry is either a predefined type or a defined class. The map is initialized with pairs key, value. Keys are binary string constants while values are *output_type* constants. If multiple values are present they are specified as aggregates surrounded by curly braces, similarly to C or C++ structures.

As an example, we have:

```
class YUVblocks {         // classes are defined in the
// next section
 unsigned int Yblocks;
 unsigned int Ublocks;
 unsigned int Vblocks;
}

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

map blocks_per_component ( YUVblocks ) {
 0b00,     {4,  1,   1},  // 4:2:0
 0b01,     {4,  2,   2},  // 4:2:2
 0b10,     {4,  4,   4}   // 4:4:4
};
```

The next rule describes the use of such a map by the declaration of a variable of type vlc.

Rule 3

type (*MapName*) *name*;

The *type* here is the return type of the map *MapName*.

Example:

```
YUVblocks(blocks_per_component) chroma_format;
```

Using the above declaration, we can access a particular value of the map using the construct chroma_format.Ublocks.

1.3.1.4 Variable Length Indirect Representation Bit Fields

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

```
class val {
 unsigned int foo;
 int  bar;
};

map sample_vlc_map ( val )  {
 0b0000.001,    {0,  5},
 0b0000.0001,   {1,  -14}
};
```

The variable-length codewords are binary strings, expressed starting with ‘0b’ or ‘0x’ format, 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 (or even variable length) representation. To allow for such exceptions, parsable type declarations are allowed for map values. This is illustrated in the following example:

```
map sample_map_with_esc (val)  {
 0b0000.001,         {0,  5},
 0b0000.0001,        {1,  -14},
 0b0000.0000.1,      {5, int(32)},
 0b0000.0000.0,      {0,  -20}
};
```

As written above, when the codeword 0b0000.0000.1 is encountered in the bitstream, then the value ‘5’ is assigned to the value of the first element (val.foo), while the following 32 bits will parsed and assigned as the value of the second element (val.bar). Note that the order is significant. Using this construct, the complete behavior of the VLC mapping is described in a concise manner in a single place.

1.3.1.5 Arithmetically Coded Bit Fields
The syntax is as follows:

Rule 4:
arith(*type*, *context_type* [, *shared_context*]) *name*;

type is the type of the coded value, *context_type* is the type of the context information kept in the coder / decoder, *shared_context* is the optional name of a context to be shared by different instances of arithmetically coded fields. If only one such field exists, its context is implicitly declared inside the coder/decoder. The same is true for more fields that are coded independently. If for readability reason one field appears in many variants of the same basic class, then each time such a field is (de)coded, the (de)coder should use one same context, referenced by the name *shared_context.*

1.3.2 Composite Data Types

1.3.2.1 Classes

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 5:
[aligned] class *object_name* [is *parent_class*] [:
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 slice: aligned unsigned int(32)
           slice_start_code=0x00000101 .. 0x000001AF {
 … // here we get vertical_size_extension, if present
 if (scalable_mode==DATA_PARTITIONING) {
      unsigned int(7) priority_breakpoint;
 }
 …
};
```

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

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

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.

1.3.2.2 Parameter types

A parameter type defines a class with parameters. This is to address cases where the data structure of the class depends on variables of one or more other objects. The concept of parameter types is consistent with the key design principle in MSDL-S, i.e. a declarative approach that allows conditional parsing in specifying syntax data structures. In fact, parameter types are a very intuitive facility, and, once defined, can be seen to be widely needed in resolving cross-references between objects.
The syntax of a class with parameters is:

Rule 6:

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

The parameter list is a list of type name/variable name pairs separated by commas. Any element of the bitstream, or value derived from the bitstream with a vlc, or constant can be passed as a parameter. Temporary variables cannot be passed as parameters.
A class that uses parameter types is dependent on the objects, whether class objects or simple variables, in its parameter list. When instantiating such a class into an object, the parameters have to be instantiated objects of their corresponding classes or types.

Example:

```
class A {
 // class body
 ...
 unsigned int(4) format;
};

class B( A a, int i ) {         // B uses parameter types
 unsigned int(i) bar;
 ...
 if( a.format == SOME_FORMAT ) {
      ...
 }
 ...
};

class C {
 int(2) I;
 A a;
 B( a, I) foo;  // instantiated parameters are required
};
```

1.3.3 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.

1.3.4 Arithmetic and Logical Expressions

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

1.3.5 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, temporary variables are provided. These are strictly of local scope to the class they are defined in. 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];
…
int I;      // this is a temporary variable
for (i=0, n=0; i<size; i++) {
 if (array[i]!=0) n++;
}

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

Variables should be used as little as possible, as they reduce the readability of the description, and complicate potential optimizations that the decoder might want to implement. They are provided so that it is possible to express various complex structures, that would otherwise be difficult to accomodate.

1.3.6 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 7:

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;
 bit(1) bar_flag;
 if (bar_flag) {
      unsigned int(8) bar;
 } else {
      int(some_vlc_table) bar;
 }
 unsigned int(32) more_foo;
};
```

Here we allow two different representations for ‘bar’, depending on the value of ‘bar_flag’. We could equally well have another entity instead of the second version (the variable length one) of ‘bar’ (another object, or another variable). Note that the use of a flag necessitates its declaration before the conditional is encountered. In addition, the type of ‘bar’ in both its possible instances has to be the same (in this case ‘int’).

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.

Rule 8:

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 9:

for (*expression1*; *expression2*; *expression3*) {
    …   // identical to a C/C++ for(;;) loop
}

Rule 10:

do {
    …   // repeat at least once, and while *condition* is true
} while (*condition*);

Rule 11:

while (*condition*) {
    …   // repeat zero or more times, and while *condition* is true
}

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.
To reduce the need for temporary variables (counters etc.), we use the following structure:

Rule 12:

repeat (*times*) {
… // repeat exactly *times* times
}

Example:

```
do {
 vlc(ac_vlc_table) ac_coeff;
} while (! [0b011]);
```

This says that 'ac_coeff' is parsed using the VLC table 'ac_vlc_table'. There is at least one 'ac_coeff', and parsing should stop when the bit string '011' is detected in the bitstream.

1.3.7 Functions

Functions are used to reduce the amount of expressions that are found in the MSDL-S description, and improve readibility and simplicity.

Rule 13:

return_type name ([type parname [, type parname]*]) {
... function body in C++/Java code ...
return(value);
}

A number of useful functions are provided as built-in into the MLSD-S specification. The capability to define new functions is reserved for Flex_2 terminals, and then only as methods of specific classes. The standard built-in functions are indicated below. Note that their definition, written in standard C/C++ code, is only provided to unambiguously define the operation of the function and does not suggest a particular implementation.

```
int nonzero ( int array[], int length ) {
 int i, n = 0;
 for( i = 0; i < length ; i++ ) {
      if( array[i] != 0 ) n++;
 }
 return( n );
}
```

```
int bitfield ( int field, int index ){
 return( field & (1<<index) );
}
```

```
int rle( int array, int length ) {
 int i = 0;
 while(i < length) {
                //while the table is not full
      unsigned int(4) e;
      unsigned int(4) l;
                //  read  value  and  length  from  the
bitstream
      int ll = l;
      while (l==15) {
           unsigned int(4) l;
           ll += l;
      }
                // maybe process the escape code
      repeat(ll) array[i++]=e;
                // pad values in the output array
 }
 return(i);
                // return the length (this is useless but
                // functions need to return a value)
}
```

The next example is how `rle` is used in the NBC audio syntax:

```
class individual_channel_stream( int window_sequence ) {
 if (window_sequence >= SHORT_DATA_SEQUENCE) {
  bit(1) sf_grouping[sf_group_size[window_sequence]];
 }
 unsigned int(8) first_scalefactor;
 int sfb_cb[number_coderbands[window_sequence] - 1];
      // this defines an array that is not taken from the
      // bitstream
 rle(sfb_cb, number_coderbands[window_sequence] - 1);
      // this function fills the sfb_cb table
      // from what is read in the bitstream
 bit(1) pulse_flag;
 if( pulse_flag ) pulse_data PulseData;
 gain_control_data GainControlData;
 scalefactor_data ScaleFactorData;
 spectral_data SpectralData;
};
```

`nonzero` calculates the number of non-zero elements contained in an array of a specified length. `bitfield` extracts the n-th bit from a bit string. `rle` extracts a run-length encoded table from the bitstream.