Workgroup:	Internet Engineering Task Force
Internet-Draft:	draft-royer-cbor-language-00
Published:	26 May 2025
Intended Status:	Informational
Expires:	27 November 2025
Author:	DM. Royer
	RiverExplorer LLC

CBOR Language Specification (CLS)

Abstract

CBOR is a wonderful over the wire format for data. This specification is for a CBOR protocol language. Currently code to send and receive CBOR streamed data of the wire has to be done by hand without tools to coordinate and aid in the development. Some libraries exist, but no protocol definition format could be found that specifically handles CBOR over the wire streams.

Many applications are built and compiled with computer languages that require that the size and type of variable be known at compile time. This specification allows an application protocol designer to specific the application requirements without regard to how CBOR encodes and decodes the data.

This specification defines a CBOR protocol definition language. This is the CBOR Language Specification (CLS). This language and the sample implementation allows a developer to define the application layer objects that can be translated into and from a CBOR data streams by independently written implementations.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 27 November 2025.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Introduction

NOTICE: This is WORK IN PROGRESS

This document specifies a CBOR Language Specification (CLS). The purpose of the CLS is to be able to define a consistent over the wire protocol definitions that transfer data between endpoints with the data encoded and decoded as specified in CBOR.

A CBOR protocol compiler is tool that reads in one or more CBOR definition files and produces code that can be used by a computer. This code is herein refereed to as generated code. And the input files are called CBOR Language Specification files, or CLS files. Within this document such files in examples use the ".cls" file name extension.

CBOR specifies how to transfer the data. And CLS specifies what will be transferred.

This specification uses C/C++ and C# in several of the examples. However a generator could be produced for almost all computer languages. As the purpose of this is to generate code to be compiled and run on a computer, some programming experience is helpful to fully understand this specification.

Many of the code examples are commented using the doxygen [DOXYGEN] style of commenting.

This specification was written in parallel with the creation of an open source CBOR language protocol compiler [CLSOpenSource]. The documentation at [CLSgendocs]. This was a way to help validate the open source compiler and this specification.

Many computer languages have built in and application defined data types that allow for more complex objects that CBOR itself defined. The solution is breakdown the more complex data type until they are objects that can be converted to and from the CBOR stream format. And then consolidating them during decoding.

For example an object might consist of a persons name with three parts, their first, middle, and last name. This CBOR language allows for an object to be created with the purpose of the generating code that sends three CBOR text string objects without the application developer having to write the I/O code for each complex data type by hand.

All CBOR data types can be specified in this language as well as more complex types built from the CBOR data types.

Given the same CLS input file, each CBOR protocol compiler generates compatible over the wire data. Allowing independent development. Application protocols can be developed and designed using this language and implementations would not need to hand code the reading or writing of the protocol details.

The goal of this language is to allow two or more cooperating applications to exchange data. It its not to guarantee that all generated code will work on all computers. For example if the protocol definition file uses 128-bit integers, then the generated code would only compile and work on systems that support 128-bit integers. This could be done if the computer supported native 128-bit integers, or by using a library that emulates larger integers. As the size of integers and other data types are not the same on all systems. Application designers may need to adjust the protocol they define to ensure that all data types they use in the protocol are supported by all intended computers. This goal allows for both flexibility, and for future systems that may have expanded data types.

The sample implementation was written using the open source ANTLR4 compiler: https://github.com/antlr/antlr4. And is much like Yacc([YACC])/Bison([BISON]) and lex([LEX])/Flex([FLEX]) grammar.

Some data requires synchronous interactions. And other data may be transmitted asynchronously. See Async and Sync Data (Section 13).

Some data such as audio and video may be streamed and not sent as blocks of data. Streamed data may be send asynchronously or synchronously See Stream Data (Section 14)

3. CLS Basics

3.1. UTC and Array Overview

The UTC value used in this specification is designed to be compatible with time_t on [POSIX] compliant systems. In POSIX systems, time_t is defined as an integer type used for representing time in seconds since the UNIX epoch, which is 00:00:00 UTC on January 1, 1970. And in this specification is 64-bit unsigned integer.

Arrays are used in a lot of data flows. String are always an array of printable characters. Some data needs to be converted to a from network byte order. Data that does not need to be converted to and from network byte order is called 'opaque' data. A PNG file is an example of opaque data. The PNG data itself is defined so that it is readable without any additional encoding.

In this CBOR Language Specification (CLS), a data type that is an array, is fixed in length, or variable in length.

A fixed array:

- A fixed array has its length surrounded by square brackets '[' and ']'.
- Each element in a fixed array is exactly the same size.
- The total size of a fixed array is calculated by multiplying the array length by the size of each element in the fixed array.
- A fixed size array can never be empty or null.

A variable sized array. A variable array is one of two types:

- It is an array of UTF-8 characters, called a string.
- Or it is an array of some non string data type.

Common to all variable array types:

- Both a string and non-string variable array may have a maximum length surrounded by angle brackets:

  '<' max-length '>'.

- Or they specify an unlimited length by having empty angle brackets, no maximum length specified. In this context 'unlimited' means not pre-defined.

  '<>'.

- Any variable array may have zero entries at run time. This is an empty array and is valid in this language specification.
- Any variable arrays may be null at run time. This is an null array and is valid in this language specification. And are transmitted as CBOR 'null' values.

Non-string variable arrays:

- Non-string variable arrays are defined by specifying their data type, and their optional maximum length.
- Variable arrays that are not string arrays have their total size calculated by multiplying the array length (which may or might not be the same as any optional maximum length) by the size of each element in the variable array. A non-string variable array size is always less than or the same as any optional maximum length.
- All elements in a non-string variable array have exactly the same size.

String variable arrays:

- A variable array of UTF-8 characters is a 'string' array. String arrays are defined using the 'string' data type.

- Variable arrays that are string arrays (a string data type) have their total size calculated by adding up the number of octets in all of the characters in the array. Which may or might not be the same as any optional maximum length. And the size (octet count) may be up to 4 times the maximum length (character count). Each UTF-8 character may be 1, 2, 3, or 4 octets in size.

  This is why a 'string' can not be a fixed size array, as you may not know the total number of octets when generating the code. And clipping off parts of a UTF-8 4, 3, or 2 octet character makes the string useless. If a string was a fixed array, then you have to describe how to pad the unused values each time, and decide on left or right padding.

string

- A string is always a variable array. If you need a fixed size character array, use a fixed array of 8 (ASCII), 16 (UTF-16), or 32 (UTF-32), values.
- A string may contain zero characters, this is an empty string.
- A string be null, in which is valid and is transmitted as a CBOR 'null'.
- A string is an array of UTF-8 printable characters. With an optional maximum length.
- A string is an array of printable characters. And each character may be 1, 2, 3, or 4 octets in size. The length specifies the number of printable characters in the string not the size of the array. This is also why a string can not be a fixed array.
- The octet count (size) of any string, could be the same as the number of characters (length), or the size could be up to 4 times larger.
- The length is the number of characters in the array.
- The size is the number of octets in the array.
- The length is always equal to or less than the size.

```
// A string variable array of printable
// characters, unlimited in length.
//
string MyVariableName<>;

// A string variable array of printable
// characters, up to 25 characters in
// length. And up to 100 octets is size.
//
string MyVariableName2<25>;
```

opaque

- An opaque array is an array of 8-bit octets. And will not be network byte encoded or decoded.
- The size and length of an opaque array are always the same value.

```
// This is a fixed array that is always
// 42 octets in size.
// All 'opaque' objects are 8-bit values.
//
opaque MyOpaqueData[42];

// A variable length array up to 42
// octets in size.
//
opaque MyOpaqueData<42>;

// A variable length array of unlimited
// number of 8-bit octets.
//
opaque MyOpaqueData<>;
```

Variable arrays in CBOR have a "<>" in them to indicate they are variable arrays.

Fixed arrays in CBOR have a "[]" in them to indicate they are fixed size arrays.

3.2. Indefinite Length Arrays

Variable arrays that have their encoding started before the length is know will transmit the data with the CBOR additional information value 31. And will be terminated with the CBOR "break" stop code 31. As defined in CBOR.

And they will be tagged in the protocol definition file as "[stream]".

No array without the "[stream]" tag will be encoded as a CBOR indefinite length value.

An array with the "[stream]" tag may be transmitted as a CBOR non-indefinite length value if known before the encoding begins.

Examples of an indefinite variable length array where the encoding starts before the length is known:

```
[stream] float16_t MicrophoneInputData<>;

[stream] uint8_t LiveTelemetry<>;

[stream] string LiveTranscription<>;
```

Figure 1

The "[stream]" tag is a hint to the implementation that the data tagged needs to be able to be processed before the length is known.

3.3. Channels

Some interactions in a data stream require a complete transfer of data before the logic of the application can proceed. This type of data is synchronous data, and is the default. It does not mean that the other endpoint must reply. It means the data must be transferred or sent before the application goes to its next programmed operation.

In some cases the data transfer can occur over time. For example a live transcription can occur at the same time that a user is reading their email on the same session. One way to implement this is with threading. Threading is a programming technique where separate flows of application logic can be processed in parallel.

There are two basic ways to do parallel data transfer.

- Open two or more sessions. One for each parallel operation.
- Multiplex the data over one connection.

Opening two or more sessions is out of scope for this specification because no additional specification beyond what is needed with multiplex coordination would be needed. And Servers may limit the number of simultaneous connections from the same authenticated user.

A multiplexed transfer of data requires a in-stream switching mechanism. This mechanism is called channeling. Both endpoints must be capable of and agree to channeling.

All objects that are to be channeled have the "[channel N]" tag applied to them. With 'N' being the channel number. The "[channel N]" tag can be added to any data type, structure, or method.

All objects that do not have a "[channel N]" tag are done in the single session connection, which is channel zero (0) and must be done synchronously.

The "[channel N]" tag has a mandatory parameter indicating the channel number or name. Channel numbers are positive integers. Channel zero (0) is reserved for synchronous communication only and is the default. Channel names must be identifiers (Section 4). And only one synchronous channel may exist, and it must be channel zero (0). When not tagged, the default is "[channel 0]" Tagging with "[channel 0]" is the same as not tagging the object.

The multiplexing is done without the application knowing it is done. The application sends a blob of data, and the underlying implementation breaks it into pieces as needed based on the amount of backlogged data across all channels.

As an aid to reading CLS files, a channel may be assigned a name. The name must be defined before its first use in "[channel N]".

Exampling naming a channel:

```
channel[1] = Notifications;

// Channel numbers only have to be
// unique positive numbers,
// They do not have to be sequential.
//
channel[12] = LiveTranscript;

// And used:
//
[channel LiveTranscript] [stream] string TranscriptText<>;

[channel Notifications] [async] GetNotifications(Callback_Identifier);
```

Figure 2

3.4. Async

All Objects may be tagged with the "[async]" tag. The "[async]" tag indicates that the application has sent the data, and will not wait for the reply. Instead, the CLS implementation will notify the application a reply has been received.

3.5. Number of bits in value

The CBOR protocol Language allows for bit value, bits values, and bit masks, See BITS (Section 12)

In this specification terminal values may specify a bit width. Indicating the number of bits in the value.

Where the "width" is the number of bits in the value. And must be an unsigned integer greater than zero. And is always expressed in decimal.

When the left side of a rule has a width: The number of bits on the left side must equal the number of bits on the right side.

The most significant values are placed to the left of lesser signification values in the rule:

In this example A Header is 32-bits in size and is composed of an 8-bit (Offset), 2-bit (Flags or F), and 22-bit (Length) value.

Here in CBOR protocol language:

```
Header:32 = Offset:8 Flags:2 Length:22
```

Figure 3

They are written most significant bits to least significant bits. So, Length (Figure 3) is 22 bits and occupies the least significant bits in the value.

Example pseudo code for the CLS in Figure 3 could be:

```
// Header is a 32-bit unsigned integer.
// Offset is an 8-bit unsigned integer.
// Flags (F) is a 2-bit unsigned integer.
// Length is a 22-bit unsigned integer
//
Header = (Offset << 24) | (Flags << 22) | Length;
```

Figure 4

The pseudo code in Figure 4 shifts the 8-bit "Offset" over 24 bits to the left, then shifts the 2-bit value "Flags (F)" over 22 bits, then, places the lower 24-bits "Length" into the results. The result would be all three values into the one 32-bit result as illustrated in Figure 5.

NOTE: Bits do not have to fill up any value, and can span multiple values. See BITS (Section 12)

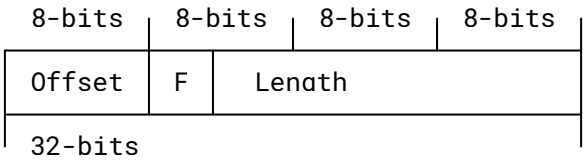

Figure 5: Packed Bit Example

3.6. Data Types - Detail

The CLS language allows the developer to code for application, many of which are built with computer languages that require known and compiled in data sizes. These sizes are converted to and from CBOR stream format as needed.

TYPE	Notes
uint_t	An unsigned integer of any size. When the CLS file specifies a uint_t, then it could be CBOR encoded as a CBOR major type 0, or as a CBOR major type 6 tag 2 bignum. It is encoded as a CBOR major type 0, unless it is larger than 64-bits.
uint8_t	An 8-bit unsigned integer. Encoded as a CBOR major type 0.
uint16_t	A 16-bit unsigned integer. Encoded as a CBOR major type 0.
uint32_t	A 32-bit unsigned integer. Encoded as a CBOR major type 0.

TYPE	Notes
uint64_t	A 64-bit unsigned integer. Encoded as a CBOR major type 0.
int_t	An signed integer of any size. When the CLS file specifies a int_t, then it could be CBOR encoded as a CBOR major type 1, or as a CBOR major type 6 tag 2 or tag 3 bignum. It is encoded as a CBOR major type 1, unless it is larger than 64-bits.
int8_t	An 8-bit signed integer. Encoded as a CBOR major type 1.
int16_t	A 16-bit signed integer. Encoded as a CBOR major type 1.
int32_t	A 32-bit signed integer. Encoded as a CBOR major type 1.
int64_t	A 64-bit signed integer. Encoded as a CBOR major type 1.
float_t	A floating point number. When the CLS file specifies a float_t, then it will be CBOR encoded as a CBOR major type 7, or major type 6 tag 5 bigfloat.
float[e.m]_t	A floating point number with a minimum exponent precision bits of 'e' bits and a minumum mantassa precission bits of 'm' bits. When the CLS file specifies a float_t, then it will be CBOR encoded as a CBOR major type 7, or major type 6 tag 5 bigfloat when too large for a type 7. Examples: float8.23_t, float11.52_t, float15.113_t IMPLEMENTATION NOTE: Most platforms and compilers do not support random bit floating point number widths. Implementors should use caution when specifying non standard bit widths in floating point numbers. Protocol designers should specify in the protocol file any size or precision requirements as comments in the file. • float5.10_t type is a float16_t. • float8.23_t type is a float32_t. • float11.52_t is a float64_t.
float16_t	A 16-bit signed integer. Encoded as a CBOR major type 7. The same as a float5.10_t type. IMPLEMENTATION NOTE: Not all platforms and compilers support 16 bit floating point numbers. Conversion methods may need to be implemented in such cases. Implementors should use caution when specifying that a 16-bit floating point should be used.
float32_t	A 32-bit IEEE floating point number. Encoded as a CBOR major type 7. The same as a float8.23_t type.
float64_t	A 64-bit IEEE floating point number. Encoded as a CBOR major type 7. The same as a float11.52_t type.

TYPE	Notes
string	A string of UTF-8 characters. Encoded as a CBOR major type 3.
opaque	An array of 8-bit values that will not be CBOR encoded or CBOR decoded when transferring the data over this protocol. Encoded as a CBOR major type 2.
map	A map is an ordered list of key/value pairs. With the key being unique in the map. A map is converted to a CBOR major type 5 map.
multimap	A multimap is an ordered list of key/value pairs. Where the keys are not necessarily unique. Duplicate keys are allowed. A multimap is converted to a CBOR major type 5 map. With each unique key in the multimap having a value part that is a CBOR major type 4 array with ordered values. One array entry for each value in in the multimap.
Op	An 8-bit value. When the highest bit is one (1) it is a vendor specific Op. Otherwise, it is set to zero (0).
true	An 1-bit value. A value of true. Set to '1'. A true value is sent as a CBOR major type 0 data item that might be packed with other bits, as defined in the CLS definition file.
false	An 1-bit value. A value of false. Set to '0'. A true value is sent as a CBOR major type 0 data item that might be packed with other bits, as defined in the CLS definition file.
enabled	A true or false value.
VENDOR_BIT	A 1 bit wide attribute value, set to 1. It is placed in the highest bit position in the value. Applies as specified in this specification. This is not a bit field, it is an attribute of the command being transmitted.
PHOENIX_BIT	A 1 bit wide attribute value, set to 0. It is placed in the highest bit position in the value. Applies as specified in this specification. This is not a bit field, it is an attribute of the command being transmitted.

Table 1: Phoenix Protocol types

Floating point numbers implementation notes. Many sizes exist depending on the operating system and compilers used. Some sizes used are 16, 24, 32, 40, 64, and 80, 96, and 128 bit floating point numbers. Generally implementations should go to the next larger size in their implementation if they do not natively support a size in the protocol definition file. Protocol designers should specify in the protocol file any size or precision requirements as comments in the file.

Implementation that require specific floating point widths could implement them as a user defined data type:

```
struct MyMadeUpFloat96
{
  bit IsNegative;
  bits Exponent:16;
  bits Mantissa:79;
};
```

Figure 6

3.7. StringRef and DataRef

This protocol allows for the reference to strings and data objects, by octet offset into the object. This is is called a StringRef or DataRef. The StringRef and DataRef do not contain the string or data, it is a reference to data in an existing string or data object. An example usage is in an index object that points to data within a string or data, such as the parts of a multipart MIME object. They consists of two parts:

Name	Size	Description
Offset	uint_t	The octet count to the start of the data. With zero being the first octet in the object.
Length	uint_t	The length in octets of the referenced object in the data.

Table 2: StringRef and DataRef ABNF/CBOR Mapping

The only difference between a StringRef and a DataRef is that a String ref points to readable string values that may be 1, 2, 3, or 4 octets in length each. And a DataRef points to objects that are always 8-bits in size.

Here is an example of a StringRef over the wire that is 8 octets in size.

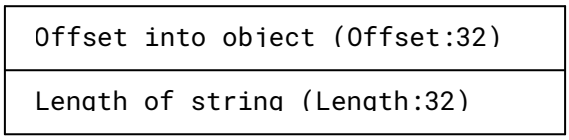

Figure 7: StringRef Format

ABNF:

3.8. CBOR Language Predefined Data Types

Here is a summary of all of the predefined data types:

This protocol language closely resembles the CBOR data model. The following data types are defined in the language. Each of these basic data types have a correlating CBOR data type. When possible the data types used in this language are the same as their [POSIX] counterpart.

Some computer languages specify the size of integers. With this language, an implementation still specifies the size and the implementation reduces the lesser over the wire size when possible. And this implementation will expand them as needed.

For example, the application may require a 64-unsigned integer (uint64_t) named CustomerCount. And assume only four (4) customers exists. So the implementation sends a one (1) octet CBOR value over the wire, in hex it would be 0x04.

That exemplifies the unique data models of the the application data size and the CBOR over the wire data size. It is the jobs of a protocol compiler to translate the received 0x01 into an uint64_t as described in this example. And to notice that a 64-bit value, depending on the size of its value, could be sent in as few as one octet to as many as ten.

3.8.1. Integers

Integers can be signed or unsigned. And they can be any size that is a multiple of 8 bits wide.

3.8.1.1. Unsigned Integer

This language specifies unsigned integers that are 8 bits, 16 bits, 32 bits , 64 bits, and bignum.

uint[W]_t
: An unsigned integer, 'W' bits wide. Where 'W' must be a multiple of 8.

  Examples of sizes not predefined: int40_t, int128_t

The ANTLR specification for unsigned integer:

```
unsignedInteger
  : 'uint_t'
  | 'uint8_t'
  | 'uint16_t'
  | 'uint32_t'
  | 'uint64_t'
  | 'uint' DECIMAL '_t'
  ;
```

Figure 8: unsignedInteger ANTLR grammar

See Section 3.8.1.1 for details.

Examples:

```
uint8_t EightBitsWide;

uint16_t SixteenBitsWide;
```

A CBOR implementation transmits the unsigned integer value, not the data type. A uint128_t unsigned integer with a value of one (1) would be transmitted as 0x01, a one octet value.

The 72, 128, and 'W' bit length unsigned integer data types are just examples used to illustrate that this protocol definition language can specify what the application needs and an implementation translates that to and from the smallest CBOR data value when used. And note that the 72 and 128 ones are valid in this language along with other nonstandard sizes.

3.8.1.2. Signed Integer

This language specifies signed integers that are 8 bits, 16 bits, 32 bits , 64 bits, and bignum.

intW_t
: An signed integer, 'W' bits wide. Where 'W' must be a multiple of 8.

ANTLR:

```
signedInteger
  : 'int_t'
  | 'int8_t'
  | 'int16_t'
  | 'int32_t'
  | 'int64_t'
  | 'int' DECIMAL '_t'
  ;
```

Figure 9: signedInteger grammar

See Section 3.8.1.1 for details.

Examples:

```
int8_t EightBitsWide;

int16_t SixteenBitsWide;
```

A CBOR implementation transmits the signed integer value, not the data type. A int128_t signed integer with a value of negative one (-1) would be transmitted as a CBOR major type 1 0b00100000 (0x20), a one octet value.

The 72, 128, and 'W' bit length signed integer data types are just examples used to illustrate that this protocol definition language can specify what the application needs and an implementation translates that to and from the smallest CBOR data value when used. And note that the 72 and 128 ones are valid in this language.

3.8.1.3. Big Integers

CBOR supports larger signed and unsigned integers as 'bignum'. An implementation conforming to this specification will transmit the value in a small of a CBOR size as possible.

A value of one (1) could be transmitted as a bignum, and it would not conform to this specification as the single octet version is smaller. The non-bignum version is to be used when possible. When they fit, unsigned integers are to be transmitted as CBOR major type 0, and signed integers as CBOR major type 1.

3.8.2. Float

ANTLR:

```
float
  : 'float16_t'
  | 'float32_t'
  | 'float64_t'
  | bigNumFloat
  ;
```

Figure 10: float grammar

Examples:

```
float16_t Size;

float32_t BiggerSize;
```

Figure 11: Float Examples

Floating point numbers are similar to unsigned integers except they are prefixed with 'float' and not 'uint'.

float16_t A 16-bit floating point number. And the corresponding CBOR data type is float.

float32_t A 32-bit floating point number. And the corresponding CBOR data type is float.

float64_t A 64-bit floating point number. And the corresponding CBOR data type is float.

float96_t A 96-bit floating point number. On some systems this is a long double. And the corresponding CBOR data type is bigfloat.

float128_t A 128-bit floating point number. On some systems this is a long double. And the corresponding CBOR data type is bigfloat.

... floatLL_t ... Assuming LL is more than 64 and a multiple of 8. An LL-bits floating point number. And the corresponding CBOR data type is bigfloat.

3.8.3. String

string A string. A A variable array of characters.

Examples:

```
// A string that has no less than 1
// and no more than 30 characters.
//
string AStringVariableName<1,30>;

// A string that has no less than 4
// and no limit on the total size.
//
string OtherString<4,*>;

// A string that has exactly 40 characters.
//
string OtherString[40];
```

Figure 12: String Examples

ANTLR:

```
string : 'string' ;
```

Figure 13: string grammar

3.8.4. Maps and MultiMaps

A map is a key and value pair, all keys in a map are unique. A multimap is a map where duplicate keys may exist.

A map is a CBOR major type 5 as defined in CBOR [RFC8949].

CBOR mandates that the keys be in bytewise lexicographic order Section 4.2.1 of [RFC8949]. Some application keys can be objects, and not simple integer or string values, making objects sorting order not easily determinable. Two things needed to be added to an object to make it a sortable key.

(1) The object defined using this language must be marked with the '[sortable]' attribute.

(2) A method must be defined in the object named 'Compare' that will be implemented by the implementer to determine the sorting order.

This Compare method returns -1 when the object is considered less than the Other object. This Compare method returns 0 when the object is considered equal to the Other object. This Compare method returns 1 when the object is considered greater than the Other object.

Here is an example of two objects that form one key (Outer):

```
[sortable]
struct InnerObject
{
  string   Name;

  uint8_t  Age;

  // A [sortable] object must have a Compare() method.
  //
  // The implementations and applications that use
  // this object must agree on how this comparison
  // is done.
  //
  int Compare(InnterObject Other);
};

[sortable]
struct OuterObject
{
  uint64_t    UniqueID;

  InnerObject NameAge;

  // In this example, this Compare method also
  // calls the NameAge.Compare() to determine
  // the sorting order. However in another case
  // it is possible it can be compared with just the
  // UniqueID.
  //
  // The implementations and applications that use
  // this object must agree on how this comparison
  // is done.
  //
  int Compare(OuterObject Other);
};

// And here the complex OuterObject is being used
// as a map key.
//
map<OuterObject,uint8_t> UsesComplexKey;
```

Each unique Compare() method defined, must be implemented by the implementer. One for each object or variable marked as '[sortable]'.

When an object it marked as sortable, and it contains one or more objects that are marked as sortable, then the outer most object's Compare() method result is the one that will be used to determine the sorting order. Which may or might not call the other Compare() methods for any included objects. How they are compared is part the the using application design.

3.8.4.1. Map

The CBOR language map data type has two arguments, the key type, and the value type. And is defined in ANTLR as:

```
map
  : 'map' '<' typeSpecifier ',' typeSpecifier '>'
  ;
```

Figure 14: map grammar

Here are examples a map in the CBOR language:

```
// The key is an 8-bit integer.
// The value is a 32-bit integer.
//
map<uint8_t,int32_t>     VariableName1;

// The key is an 8-bit integer.
// The value is a string with at least one character.
//
map<uint8_t,string<1,*>> VariableName2;

// Two example objects.
//
struct Object1
{
  int32_t   Collected[45];
  float16_t Values[45];
};

struct Object2
{
  Object1   CurrentItems<>;
  Object1   History<>;
};

// A more complex map.
//
map<Object1,Object2>   SomeVariableName3;
```

3.8.4.2. Multimap

A multimap allows for duplicate keys.

When an application has a multimap that needs to be transmitted via CBOR, it sends a map, except the value becomes an array of objects, all that have the same key. One key sent, all of the objects with that key sent as an array, as the value to that key.

Here is the ANTLR definition of a multimap in the CBOR language:

```
multimap
  : 'multimap' '<' typeSpecifier ',' typeSpecifier '>'
  ;
```

Figure 15: multimap grammar

A multimap is a map that allows multiple of the same keys. Here is an example of a multimap of a graphic object being duplicated at multiple positions in a 3 dimensional space.

The multimap uses the same sorting as described in Maps (Section 3.8.4.1)

```
struct Vector3
{
  float32_t X;
  float32_t Y;
  float32_t Z;
};

// An example of a multimap where the key is the identifier of
// a specific graphic object type, with an ID that is an
// unsigned 32-bit integer.
//
// And the value is its position.
//
// Any object may have multiple instances of the same graphic
// object, all with the same ID, and each with its position
// at a different location.
//
multimap<uint32_t,Vector3> Instances;
```

This example would be transmitted as one CBOR map. There would be one map key for each unique uint32_t key. And Each key would have an array of one or more Vector3 objects.

The protocol compiler would implement; for each unique multimap, a method to convert and encode a computer language multimap into a CBOR map. And a method to decode and convert the map back into a multimap.

3.9. Arrays

Arrays can be fixed length, or variable length. Fixed length arrays have all of their elements transmitted each time, thus each element must contain a valid value.

3.9.1. Fixed Length Arrays

Fixed length arrays have their size bounded by square brackets. Here are some examples:

Variable Length Array Examples.

```
// An array that consists of three 32 bit floating
// point numbers.
//
float32_t Transform3D[3];

// An array that contains 1024 8-bit signed integer values.
//
int8_t Samples[1024];
```

Figure 16: Arrays Examples

3.9.2. Variable Length Arrays

Variable length arrays can have a variable number of elements. You can specify the minimum number of elements, the default is zero (0). You can specify the maximum number of elements, the default is an unlimited amount of entries. An unlimited maximum value is indicated with the '*' symbol.

Fixed length arrays have their size bounded by angle brackets. Here are some examples:

Variable Length Array Examples:

```
// A variable length array that can hold zero,
// or up to an unlimited number of 32 bit
// floating point entries.
// The minimum value is empty and the maximum value is empty.
//
float32_t Samples<>;<

// A variable length array that can hold zero,
// or up to 100 32-bit unsigned entries.
//
uint32_t DistanceMarks<0,100>;

// A variable length array that must contain at least 2
// 32-bit floating point numbers and no more than
// 128 entries.
//
float32_t Trajectory<2,128>;

// A variable length array that must contain at least 2
// 32-bit floating point numbers and an unlimited
// number of entries.
//
float32_t Variation<2,*>;
```

4. Identifiers and Variables

Each item encoded and decoded in an application will be paced into a memory location of the application at run time. These places are named with identifiers.

And the generic name for this kind of identifier is 'variable'. And a variable may contain any kind of data the applications needs. These memory locations are named by the application developer and are called variables.

Another usage of an identifier is to define a new data type to be used by the application.

All variables have a data type (integer, float, ...) and are named in this language specification. And there are user defined data types that build from the built in data types and other user defined data types.

4.1. Identifiers and Scope

Not all protocols will need to scope variables. The primary reason for scoping is to reduce name collisions in larger and more complex protocols.

An identifier can be public for any code to use. This is in the 'public' scope.

Or they can be internal to the objects being described. This is in the 'internal' scope. For example, a protocol might also define a method to validate the data being provided. This validation method might be scoped as internal so that access from other objects and public access is not possible. In such cases the protocol is defining that the implementer must implement the defined validation method.

Or they can only be accessible within a 'namespace'. This is in the 'namespace' scope. A namespace is a way of identifying a collection of protocol names and definitions that may have names that are the same as the names used in a different namespace. For example a "Get()" method may exist in a MIME namespace "MIME:Get()", and in a file namespace "FILE:Get()". The namespace of protocol definitions, methods, and variables default to any namespace they are defined into, and my be defined and used by their full namespace path. See Namespaces (Section 7).

4.1.1. Publicly scoped Identifiers

Public identifiers names must start with a letter from the UTF-8 set: a-z or A-Z. And followed by zero or more from the set: a-z, A-Z, 0-9, and underscore '_'. Shown here as a POSIX regular expression for an identifier in the public scope: [a-z][A-Z]*

And the ANTLR definition. Where 'identifier' is any valid identifier, and 'IDENTIFIER' is a public identifier.

```
identifier
    : IDENTIFIER
    | internal_identifier
    ;

IDENTIFIER
  : [a-zA-Z] ([a-zA-Z0-9_])*
  ;
```

Figure 17: identifier grammar

In the example Figure 18, Speed, TimeNow, Time9, Travel_Time, Collection, FirstItem and AValue are public identifiers. As is the method name AMethodThatDoesSomething.

int32_t, uint64_t, and float32_t are built in data types.

Collection is a public identifier that is defining the name of a new user defined data type.

Examples:

```
int32_t Speed;         // Okay public, follows the rules.

UTC_t TimeNow;         // Okay public, follows the rules.

float32_t Time9;       // Okay public, follows the rules.

struct Collection      // 'Collection' is valid, and public.
{
  int8_t    FirstItem; // Okay, value is public.

  float32_t AValue;    // Okay, value is public.

  /**
   * A public method that has access to all variables.
   *
   * @note 'void' means returns nothing.
   */
  void AMethodThatDoesSomething();
};
```

Figure 18: Sample with Public Scope

4.1.2. Internal Scoped Identifiers

An internally scoped identifier is an identifier that starts with an underscore '_'.

Internally scoped identifiers can only be accessed from within the object they are defined in. They are often helper variables or methods.

Internally scoped identifiers when not in a namespace and not in a struct, are only accessible within the generated code.

Internally scoped identifiers in a namespace and not in a struct are namespace scoped. And are accessible anywhere in the same namespace.

Internally scoped identifiers inside of a struct, are only accessible within the struct.

The ANTLR for an internal identifier, with 'identifier' shown again, and 'internal_identifier' also included.

```
identifier
    : IDENTIFIER
    | internal_identifier
    ;

internal_identifier : '_' IDENTIFIER ;
```

Figure 19: internal_identifier grammar

Examples:

```
// Example:
//
// _NextSequenceNumber is internal can only be accessed in
// the generated code.
//
uint64_t _NextSequenceNumber;

// A public identifier, that has internal members.
//
struct Collection
{
  int8_t    ADevice; // Okay, value is public.

  float32_t AValue;  // Okay, value is public.

  string    _AName<>;// Okay, value is only
                     // available inside this object.

  /**
   * A public method that has access to all variables
   * including internal ones (_AName).
   * That calls _ValidateName().
   *
   * @param AName A string that could be null or any length.
   *
   * @note 'void' means returns nothing.
   */
  void SetName(string AName<>);

  /**
   * A public method that has access to all variables
   * including internal ones (_AName).
   *
   * @note Calls _ValidateName().
   *
   * @return A string that may be null, or any length.
   */
  string<> GetName();

  /**
   *
   * An internal method that has access to all variables
   * including internal ones (_AName) in 'Collection'
   * This method is only available to be called from within
   * this 'Collection' object.
   *
   * @param AName A string that may be null, or any length.
   *
   * @return true if AName is valid, or false if AName is
   * not valid.
   */
  bool _ValidateName(string AName<>);
};
```

Figure 20: Internal Scope Example

See Section 10 for struct details.

4.1.3. Namespace Scoped Identifiers

A namespace scoped identifier is an identifier that starts with an underscore '_'. And exists inside of a 'namespace' and not within a struct. See Section 7 to understand namespaces and Structs (Section 10) to understand 'struct'.

Namespace scoped identifiers can only be accessed from within the same namespace they are defined in. They are often variables and methods that aid in the cross coordination of objects within the namespace.

- An identifier defined inside of a struct that starts with an underscore '_' is an identifier that is only known within the struct and is an internally scoped identifier.
- An identifier that is in a namespace, and not within a struct is a namespace scoped identifier.

```
namespace_identifier : '_' identifier ;
```

Figure 21: namespace grammar

Examples:

```
// A public Namespace called DailyData
//
namespace DailyData
{
  /**
   * This is a namespace scoped identifier.
   * It is in a namespace scoped, and not in a struct.
   */
  uint64_t _TotalBytesSent;

  /**
   * Okay another namespace scoped identifier.
   */
  uint64_t _Travel_Time;

  /**
   * This struct is inside of a namespace.
   * And its identifier starts with an underscore '_'.
   * So _Summary is a namespace identifiers.
   * Only code in this same namespace has access
   * to this object type.
   */
  struct _Summary
  {
    int8_t    FirstItem; // Okay, value is public.

    float32_t AValue;    // Okay, value is public.

    int32_t   _AName;    // Okay, value is only
                         // available inside this struct.

    /**
     * A public method that has access to all variables
     * including internal ones (_AName).
     * And only available inside this struct
     * and namespace.
     *
     * @note 'void' means returns nothing.
     */
    void AMethod();

    /**
     * An internal method that has access to all variables
     * including internal ones (_AName).
     * This method is only available to be called from within
     * in the _Summary object.
     *
     * @note 'void' means returns nothing.
     */
    void _AnotherMethod();
};
```

Figure 22: Namespace Scope Example

See Section 10 for struct details.

5. Variables and Arrays

Variables may be non-array, fixed size array, or variable sized array:

5.1. Non-Array Variables

A non-array variable is one that is not a fixed or variable sized array. Examples:

NOTE: A 'string' is always an array.

Examples:

```
uint8_t  SimpleUnsignedInteger;
float32_t  Simple32BitFloat;
map<uint8_t,AnObjectType>; SimpleMap;
multimap<uint8_t,AnObjectType> SimpleMultiMap;
```

Figure 23: Variable No Arrays Example

5.2. Fixed Size Array '[]'

A fixed size array variable has a size enclosed in square brackets '[]'. Examples:

Examples:

```
uint8_t  FixedUnsignedInteger[30];
float32_t  Fixed32BitFloat[12];
map<uint8_t,AnObjectType>; FixedMap[34];
multimap<uint8_t,AnObjectType> FixedMultiMap[16];
string LargeString[2048];
```

Figure 24: Variable Fixed Size Array Examples

5.3. Variable Size Array '<Min,Max>'

A variable array size array has a minimum size. The default minimum size is zero.

A variable array has a maximum size. The default maximum size is unlimited.

The minimum and maximum values are separated by the comma (,) character.

An variable array size is specified in one of 3 ways:

5.3.1. Variable Size Array With Minimum and Maximum Size

<Min,Max>
Where:

Min Is the minimum size for the array. The default minimum is zero.

Max Is the maximum size of the array. The default minimum size is unlimited. Unlimited may also be specified as '*'.

Examples:

```
// An array of 5 to 14 int8_t data elements.
//
int8_t VariableName<5,14>;

// An array of 3 to 16 float32_t data elements.
//
float32_t FloatArray<3,16>;

// An string of 5 to 42 characters.
// The size would range from 5 to 168 octets in size.
//
string StringVariable<5,42>;

// An string of 8 to unlimited characters.
// The size would be from 8 to unlimited size.
//
string AString<8,*>;
```

Figure 25: Variable Variable Size Array Examples

5.3.2. Variable Size Array With Maximum Size

<Max> A shorthand for <0,Max>. Specifies a zero to Max sized array.

Example:

```
// An array of length zero to 17 characters in length.
// And a size up to 68 octets.
//
string VariableName<17>;
```

Figure 26: Variable Size Array Max Size Examples

5.3.3. Variable Size Array With Default Size

<>
A shorthand for <0,*>.

Example:

```
uint64_t VariableName<>;
```

Figure 27: Variable Array Defaults Examples

5.3.4. Variable Array Examples

Examples:

```
// A variable array that holds no more than
// 30 uint8_t items.
//
uint8_t  VariableUnsignedInteger1<30>;

// A variable array of unlimited uint8_t items.
//
uint8_t  VariableUnsignedInteger2<>;

// A variable array of unlimited uint8_t items.
//
uint8_t  VariableUnsignedInteger3<*>;

// A variable array that hold up to 12 float32_t items.
//
float32_t  Variable32BitFloat<12>;

// A more complex example.
//
// A variable array that holds up to 34 key/value maps.
//
// Each map has keys of type of uint8_t and values
// of each map are of type AnObjectType.
//
// Each of the up to 34 maps, must have a unique key to itself
// but not to the other maps.
//
map<uint8_t,AnObjectType> VariableMap<34>;

// A variable array of multimaps.
//
// Each multimaps has keys of type uin8_t and
// each multimap has values of type OtherObjectType.
//
// The array holds up to 16 of these multimaps.
// Each multimap may have duplicate keys.
//
multimap<uint8_t,OtherObjectType> VariableMultiMap<16>;
```

Figure 28: Variable Array Examples

6. Declaring Variables

A variable has a type and a name. And the name can have some attributes. The ANTLR for a variable declaration is:

```
declaration : typeSpecifier identifier
  | typeSpecifier identifier '[' value ']'
  | typeSpecifier identifier '<' value? '>'
  | 'opaque' identifier '[' value ']'
  | 'opaque' identifier '<' value? '>'
  | 'string' identifier '<' value? '>'
  | 'void'
  ;
```

Figure 29: declaration grammar

7. Namespaces

With large protocols it can be difficult to make sure that names of items are unique. Namespace allows groups of names to be encapsulated and identified without regard to the names of items in other namespaces or globally.

For example, a protocol might have a unique identifier for the sequence of an item and declare a type called UID.

Another set of exchanges might be to tranfer sets of data, each with a unique identifier and declare a type called UID.

In this simple example, it is concevable that you just make sure that you do not use both in the same set. However with large projects cross identifier errors can happen and can be difficult to track down.

Namespace solve this problem by wrapping objects, or sets of object in a 'namespace' type. Here is a simple example:

```
// Declare a type called UID that is
// a 64-bit unsigned integer.
// In this scope is a sequence number.
//
typedef unint64_t UID;

struct Packet
{
  UID Sequence;
  Blob ABlobOdData; // Blob not defined in this example.
};
```

And in another definition file you might have defined this:

```
// Declare a type called UID that is
// a 64-bit unsigned integer.
// In this scope, the UID identifies a set of data (Blob2).
//
typedef unint64_t UID;

struct EnumeratedItems
{
  UID ItemIdentifier;
        Blob2 TheItemData;// Blob2 not defined in this example.
};
```

And when you compile your protocol you get an error that you defined UID twice (or more).

The solution could be to rename one of them. Or to use namespaces:

```
namespace Packets
{
  // Declare a type called UID that is
  // a 64-bit unsigned integer.
  // In this scope is a sequence number.
  //
  typedef unint64_t UID;

  struct Packet
  {
    UID Sequence;
    Blob ABlobOdData; // Blob not defined in this example.
  };
} // End of namespace /Packets'.

namespace ItemList
{
  // Declare a type called UID that is
  // a 64-bit unsigned integer.
  // In this scope, the UID identifies a set of data (Blob2).
  //
  typedef unint64_t UID;

  struct EnumeratedItems
  {
    UID ItemIdentifie;
    Blob2 TheItemData;// Blob2 not defined in this example.
  };
}
```

When in the implementation where you need to access the public identifiers from other namespaces they could be identified by using their full scope. In C++ this could look like:

```
ItemList::EnumeratedItems Items;
Packets::Packet           APacket

ItemList::UID             ItemUID;
Packets::UID              PacketUID;
```

And in C# it would look like:

```
ItemList.EnumeratedItems Items;
Packets.Packet           APacket;

ItemList.UID             ItemUID;
Packets.UID              PacketUID;
```

8. User Type Definitions

A typedef is both a shorthand way of specifying a data type. And it allows for the identification of the intended usage of otherwise ambiguous data type usage. This can also help find coding errors.

The typedef and its concept are the same as they are in C/C++.

The ANTLR for typedef:

```
typeDef
    : 'typedef' declaration ';'
    | 'enum' identifier enumBody ';'
    | structTypeSpec ';'
//    | 'union' identifier unionBody ';'
    ;
```

Figure 30

In Figure 31 and Figure 32 there is an error as both the Sequence and Command are different sizes. If one definition file has:

```
struct Packet
{
  // A map of Sequence to command.
  //
  map<uint32_t,uint32_t> History;
  ...
};
```

Figure 31

And a second file has a definition for a method to use the packet data:

```
// On many systems this would compile
// because the 'add' line would not know
// that Sequence and command were too small as
// the compilers automatically upcast to larger sizes.
// Only later at runtime when Sequence or Command exceeded
// the size of uint16_t might the errors be caught.
//
void ProcessPacket(uint16_t Sequence, uint16_t Command)
{
  ...
  // Error - should be (Seq, Cmd)
  // However it would compile as the compiler
  // only sees that they are both unsigned integers.
  // (NOTE: map does not have a '.add', just an demonstration).
  //
  History.add(Cmd, Seq);
  ...
}
```

Figure 32

If typedefs had been used it would catch that error and also make it easier to read:

```
// Create a data type called 'PacketSequence'
//
typdef uint32_t PacketSequence;

// Create a data type called 'PacketCommand'
//
typedef uint32_t PacketCommand;

// Create a data type called 'SequenceHistory'
//
typedef map<PacketSequence,PacketCommand> SequenceHistory;

struct Packet
{
  // A map of Sequence to command.
  //
  SequenceHistory History;
  ...
};
```

Figure 33

And if the second usage had been defined as:

```
void ProcessPacket(PacketSequence Seq, PacketCommand Cmd)
{
  // This would not compile as the types do not match.
  // (NOTE: '.add' is made up for demonstartion only)
  //
  History.add(Cmd, Seq); // Error - should be (Seq, Cmd)

  // This would, and should compile.
  //
  History.add(Seq,Cmd);
  ...
}
```

Figure 34

9. Enum

Much like an enum in C/C++/C# the purpose of the enum is to create identifiers and assign numeric values to them in order to simply reading and understanding the code.

The enum ANTLR:

```
enum       = 'enum' enum-name '{'
             enum-item *(',' enum-item)
           '};'

enum-item  = UserDefinedIdentifier '=' Value

enum-name  = UserDefinedIdentifier

Value      = 0x00-0xffffffff
```

- No two enum-item identifiers may be the same in the same enum.
- No two enum-item identifiers may have the same value in the same enum.
- All enum-items are at the same scope as the enum-name
- An enum-name and therefore the enum may be in the public, internal, or namespace scope.

Examples:

```
enum RoomsInAHome {
  Kitchen = 1,
  LivingRoom = 2,
  BedRoom = 3
};

enum DaysOfTheWeek {
  Monday = 1,
  Tuesday = 2,
  Wednesday = 3,
  Thursday = 4,
  Friday = 5,
  Saturday = 6
  Sunday = 7
};
```

10. Structues

A struct is a way to encapsulate a related set of variables.

Struct ANTLR:

```
structTypeSpec
    : ('struct'|'class') identifier structBody
    ;
```

Figure 35

11. Union

todo

```
this is a test
```

12. Bit, bits, and bitmasks

13. Async and Sync Data

Some or all of a protocol can be transmitted asynchronously or synchronously.

TODO

14. Stream Data

Data that has has a variable amount of data, like mp3 audio and mp4 video are transmitted as stream data.

Stream data may be transmitted asynchronously or synchronously.

TODO

15. Program

todo

```
this is a test
```

15.1. Versioning the Protocol

todo

16. IANA Considerations

NOTE: This will be filled in with instructions and procedures to expand capabilities and commands.

17. Security Considerations

Robust digital certificate control. Especially with AUTHCERT_TLS.

MUCH MORE TODO ...

18. References

18.1. Normative References

[POSIX] IEEE, "1003.1-2024 - IEEE/Open Group Standard for Information Technology--Portable Operating System Interface (POSIX™) Base Specifications", June 2024, <https://ieeexplore.ieee.org/servlet/opac?punumber=10555527>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

18.2. Informative References

[BISON] "Bison - A Parser Generator".

[CBORImplementation] Royer, D., "CBORGEN Sample Implementation", 2025, <https://github.com/RiverExplorer/CBOR>.

[CLSgendocs] Royer, DM., "cborgen Protocol Compiler", April 2025, <https://github.com/RiverExplorer/CBOR>.

[CLSOpenSource] Royer, DM., "cborgen Open Source Tool", April 1983, <https://github.com/RiverExplorer/CBOR>.

[DOXYGEN] "DOXYGEN - Code Documentation Automated", <https://www.doxygen.nl/manual/commands.html>.

[FLEX] "Flex - A fast Lexical Analyzer".

[LEX] "Lex - A Lexical Analyzer".

[YACC] Johnson, SCJ., "Yacc (Yet Another Compiler-Compiler".

Appendix A. Complete ANGLR grammar

```
# The CBOR Phoenix cborgen grammar
#
grammar cbor

# ANTLR/bigNumFloat.g4
#
bigNumFloat : 'float' decimal '_t' ;

# ANTLR/bigNumInt.g4
#
bigNumInt : 'int' decimal '_t' ;

# ANTLR/bigNumUInt.g4
#
bigNumUInt : 'uint' decimal '_t' ;

# ANTLR/bitmaskBody.g4
#
bitmaskBody
    : '{' (identifier '=' value) (',' identifier '=' value)* '}'
    ;

# ANTLR/bitmask.g4
#
bitmaskTypeSpec
    : 'bitmask' bitmaskBody
    ;

# ANTLR/bits.g4
#
bitsTypeSpec
  : 'bit' typeSpecifier
  | 'bits' typeSpecifier width
  ;

# ANTLR/caseSpec.g4
#
caseSpec
    : 'case' value ':' declaration ';'
    ;

# ANTLR/cborSpecification.g4
#
cborSpecification
    : specs+
    ;

# ANTLR/comment.g4
#
comment : CommentOneLine
  | CommentMultiLine ;

# ANTLR/commentMultiline.g4
#
CommentMultiLine : '/*' .*? '*/' ;
```

```
# ANTLR/commentOneLine.g4
#
CommentOneLine : '//' ~[\r\n]+ ;

# ANTLR/constantDef.g4
#
constantDef
    : 'const' typeSpecifier identifier '=' constant ';'
    ;

# ANTLR/constant.g4
#
constant
    : decimal
    | float
    | hexadecimal
    | octal
    ;

# ANTLR/dateType.g4
#
dataType : typeSpecifier
  | typeSpecifier '[' value ']'
  | typeSpecifier '<' value? '>'
  | 'opaque' '[' value ']'
  | 'opaque' '<' value? '>'
  | 'string' '<' value? '>'
  | 'void'
  ;

# ANTLR/decimal.g4
#
decimal
    : DECIMAL width?
    ;

# ANTLR/DECIMAL.g4
#
DECIMAL
    : ('-')? ([0-9])+
    ;

# ANTLR/declaration.g4
#
declaration : typeSpecifier identifier
  | typeSpecifier identifier '[' value ']'
  | typeSpecifier identifier '<' value? '>'
  | 'opaque' identifier '[' value ']'
  | 'opaque' identifier '<' value? '>'
  | 'string' identifier '<' value? '>'
  | 'void'
  ;

# ANTLR/definition.g4
#
definition
    : typeDef
    | constantDef
```

```
    ;

# ANTLR/enumBody.g4
#
enumBody
    : '{' (identifier '=' value) (',' identifier '=' value)* '}'
    ;

# ANTLR/enumTypeSpec.g4
#
enumTypeSpec
    : 'enum' identifier enumBody
    ;

# ANTLR/float.g4
#
float
  : 'float16_t'
  | 'float32_t'
  | 'float64_t'
  | bigNumFloat
  ;

# ANTLR/FLOAT.g4
#
FLOAT
    : ('-')? ([0-9])+('.' [0-9+]*)? (('e'|'E')('-')?[0-9]+)?
    ;

# ANTLR/floatValue.g4
#
floatValue
    : FLOAT
    ;

# ANTLR/hexadecimal.g4
#
hexadecimal
    : HEXADECIMAL width?
    ;

# ANTLR/HEXADECIMAL.g4
#
HEXADECIMAL
    : '0x' ([a-fA-F0-9])+
    ;

# ANTLR/identifier.g4
#
identifier
    : IDENTIFIER
    | internal_identifier
    ;

# ANTLR/IDENTIFIER.g4
#
IDENTIFIER
  : [a-zA-Z] ([a-zA-Z0-9_])*
```

```
  ;

# ANTLR/ignoreTag.g4
#
ignoreTag : '[ignore]'
          ;

# ANTLR/internal_identifier.g4
#
internal_identifier : '_' IDENTIFIER ;

# ANTLR/internalTag.g4
#
internalTag : '[internal]'
          ;

# ANTLR/map.g4
#
map
  : 'map' '<' typeSpecifier ',' typeSpecifier '>'
  ;

# ANTLR/method.g4
#
method: dataType identifier '(' dataType identifier?
        (',' dataType identifier?)* ')' ';'
  ;

# ANTLR/multimap.g4
#
multimap
  : 'multimap' '<' typeSpecifier ',' typeSpecifier '>'
  ;

# ANTLR/naemspaceDef.g4
#
namespaceDef
  : 'namespace' identifier ( ':' identifier )* ';'
  ;

# ANTLR/namespace_identifier.g4
#
namespace_identifier : '_' identifier ;

# ANTLR/namespaceTag.g4
#
namespaceTag : '[namespace]'
             ;
# ANTLR/octal.g4
#
octal : OCTAL width?
      ;

# ANTLR/OCTAL.g4
#
OCTAL
    : '0' [1-7] ([0-7])*
    ;
```

```
# ANTLR/overrideTag.g4
#
overrideTag : '[override]'
            ;

# ANTLR/PASS.g4
#
PASS
  : '%' ~[\n\r]+
  | '%' [\n\r]+
  ;

# ANTLR/passThrough.g4
#
passThrough : PASS
   ;

# ANTLR/privateTag.g4
#
privateTag : '[private]'
           ;

# ANTLR/procFirstArg.g4
#
procFirstArg
  : 'void'
  | dataType identifier?
  ;

# ANTLR/program.g4
#
program : 'program' identifier '{' version+ '}' '=' value ';'
  ;

# ANTLR/publicTag.g4
#
publicTag : '[public]'
          ;

# ANTLR/QIDENTIFIERDOUBLE.g4
#
# A value enclosed in double quotes (")
#
QIDENTIFIERDOUBLE
  : '"' [a-zA-Z] ([a-zA-Z0-9_])* '"'
  ;


# ANTLR/QIDENTIFIERSINGLE.g4
#
QIDENTIFIERSINGLE
  : '\'' [a-zA-Z] ([a-zA-Z0-9_])* '\''
  ;
# ANTLR/secs.g4
#
specs
  : structTypeSpec
```

```
  | unionTypeSpec
  | enumTypeSpec
  | constantDef
  | namespaceDef
  | comment
  | passThrough
  | program
  | declaration ';'
  | typeDef
  ;

# ANTLR/signedInteger.g4
#
signedInteger
  : 'int_t'
  | 'int8_t'
  | 'int16_t'
  | 'int32_t'
  | 'int64_t'
  | 'int' DECIMAL '_t'
  ;


# ANTLR/sortableTag.g4
#
sortableTag : '[sortable]' ;

# ANTLR/string.g4
#
string : 'string' ;

# ANTLR/structBody.g4
#
structBody
    : '{'
  ((declaration ';') | comment+ | method+ | passThrough+)
  ((declaration ';') | comment+ | method+ | passThrough+)*
  '}'
    ;

# ANTLR/structTypeDef.g4
#
structTypeSpec
    : ('struct'|'class') identifier structBody
    ;

# ANTLR/tags.g4
#
tags : ignoreTag
     | internalTag
     | overrideTag
     | privateTag
     | namespaceTag
     | sortableTag
     ;

# ANTLR/typeDef.g4
#
```

```
typeDef
    : 'typedef' declaration ';'
    | 'enum' identifier enumBody ';'
    | structTypeSpec ';'
//    | 'union' identifier unionBody ';'
    ;

# ANTLR/typeSpecifier.g4
#
typeSpecifier
  : bitsTypeSpec
  | bitmaskTypeSpec
  | 'bool'
  | enumTypeSpec
  | float
  | map
  | multimap
  | 'opaque'
  | signedInteger
  | 'string'
//  | structTypeSpec
//  | unionTypeSpec
  | unsignedInteger
  | identifier
  ;

# ANTLR/unionBody.g4
#
unionBody
    : caseSpec
    | comment
    | passThrough
    | 'default' ':' declaration ';'
    ;

# ANTLR/unionTypeSpec.g4
#
unionTypeSpec
  : 'union' identifier 'switch' '(' declaration ')'
    '{'
       unionBody+
    '}' ';'
  ;

# ANTLR/unsignedInteger.g4
#
unsignedInteger
  : 'uint_t'
  | 'uint8_t'
  | 'uint16_t'
  | 'uint32_t'
  | 'uint64_t'
  | 'uint' DECIMAL '_t'
  ;


# ANTLR/value.g4
#
```

```
value
    : constant
    | identifier
    | QIDENTIFIERSINGLE
    | QIDENTIFIERDOUBLE
    ;

# ANTLR/version.g4
#
version : 'version' identifier '{' versionMethod+ '}' '=' value ';'
  ;

# ANTLR/versionMethod.g4
#
versionMethod: dataType identifier
 '(' ((dataType identifier?
 (',' dataType identifier?)*) | 'void') ')' '=' value ';'
  ;

# ANTLR/width.g4
#
width
  : (':' DECIMAL)
  ;

# ANTLR/WS.g4
#
WS
    : [ \t\r\n]+ -> skip
    ;
```

Figure 36

Acknowledgments

Contributors

Thanks to all of the contributors. [REPLACE]

Author's Address

Doug Royer
RiverExplorer LLC
848 N. Rainbow Blvd #1120
Las Vegas, Nevada 89107
United States of America
Phone: 1+208-806-1358
Email: DouglasRoyer@gmail.com
URI: https://DougRoyer.US