
Workgroup:	Internet Engineering Task Force
Internet-Draft:	draft-royer-cbor-language-00
Published:	3 May 2025
Intended Status:	Informational
Expires:	4 November 2025
Author:	DM. Royer <i>RiverExplorer LLC</i>

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.

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 4 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	3
2. Introduction	4
3. CLS Type	5
3.1. CLS Type	5
3.2. Number of bits in value	9
3.3. Common Definitions	10
3.4. StringRef	11
4. Identifiers and Variables	12
4.1. How to name a variable or identifier	12
4.1.1. Publicly scoped Identifiers	12
4.1.2. Internal Scoped Identifiers	13
4.1.3. Namespace Scoped Identifiers	14
5. Variables and Arrays	16
5.1. Non-Array Variables	16
5.2. Fixed Size Array '[]'	16
5.3. Variable Size Array '<Max>'	16
5.3.1. Variable Size Array With Minimum and Maximum Size	17
5.3.2. Variable Size Array With Maximum Size	17
5.3.3. Variable Size Array With Default Size	17
5.3.4. Variable Array Examples	18
6. CBOR Language Predefined Data Types	19
6.1. Integers	20
6.1.1. Unsigned Integer	20
6.1.2. Signed Integer	21
6.1.3. Big Integers	22
6.2. Float	22

6.3. String	23
6.4. Maps and MultiMaps	24
6.4.1. Map	25
6.4.2. Multimap	26
7. Arrays	27
7.1. Fixed Length Arrays	27
7.2. Variable Length Arrays	28
8. Declaring Variables	28
9. Namespaces	29
10. User Type Definitions	31
11. Enum	33
12. Structues	34
13. Union	34
14. Program	34
14.1. Versioning the Protocol	34
15. IANA Considerations	35
16. Security Considerations	35
17. References	35
17.1. Normative References	35
17.2. Informative References	35
Appendix A. Complete ANGLR grammar	35
Acknowledgments	39
Contributors	40
Author's Address	40

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.

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 compled data type until they are objects that can be converted to and from the CBOR stream format.

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/Bison and lex/Flex grammar.

3. CLS Type

3.1. CLS Type

The meaning of "CLS TYPE" in this specification refers to a data type as a "C" code API type. The "C" language was chosen because many other languages have methods of calling and using "C" APIs methods and data.

For every "CLS TYPE" "foo", is a "C" code API of "cbor_foo(...)" that can encode and decode the data to and from a CBOR over the wire value. The "CLS TYPE" is normative. The "C" API is informative and meant as a guide to implementors.

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, 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.

NAME	Description:
string	<ul style="list-style-type: none">• 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. <pre>// 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>;</pre>

NAME	Description:
opaque	<ul style="list-style-type: none">• 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. <div><pre>// 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<>;</pre></div>

Table 1: CLC strings and arrays.

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

```
CBOR:  
string StringOfCharacters<>;  
Widget ArrayOfWidget<>;  
opaque PNGFileData<>;
```

Figure 1

Where:

- sting: A CBOR protocol data type.
- StingOfCharacters A user defined name for this array.
- Widget: An example of a a user defined data type.
- Array of Widget: A user defined name for this array.
- opaque: A CBOR protocol data type.
- PNFFFileData: A user defined name for this array.

The opaque data type represents data that is not encoded in any way. A PNG file is an example that does not get network byte order translated, it is sent as an opaque blob of unaltered data.

A 'string' in CBOR represents a string of printable characters.

3.2. Number of bits in value

The CBOR protocol allows for bit value, and streams bits.

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.

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

Figure 2

Example pseudo code for the ABNF in [Figure 2](#) 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 3

The pseudo code in [Figure 3](#) 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 4](#):

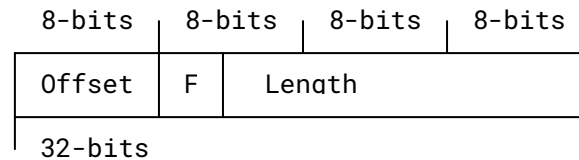


Figure 4: Packed Bit Example

3.3. Common Definitions

TYPE	Notes	CBOR API
uint8_t	An 8-bit unsigned integer.	cbor_uint8_t()
uint16_t	A 16-bit unsigned integer.	cbor_uint16_t()
uint32_t	A 32-bit unsigned integer.	cbor_uint32_t()
uint64_t	A 64-bit unsigned integer.	cbor_uint64_t()
string	A string of UTF-8 characters.	cbor_string()
opaque	An array of 8-bit values that will not be CBOR encoded or CBOR decoded when transferring the data over this protocol.	cbor_opaque()
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).	cbor_Op()
OpSet	An 8-bit value. Signifies the operation will set a value. This is used in a few places to signify the operation to apply to any included data.	cbor_Op() With the value cat to a (Op).
OpGet	An 8-bit value. Signifies the operation will get a value.	cbor_Op() With the value cat to a (Op).
OpUpdate	An 8-bit value. Signifies the operation will update an existing value.	cbor_Op() With the value cat to a (Op).
OpDelete	An 8-bit value. Signifies the operation will delete key/value pair.	cbor_Op() With the value cat to a (Op).
true	An 8-bit value. A value of true.	cbor_false()
false	An 8-bit value. A value of false.	cbor_true()
enabled	A true or false value.	cbor_enabled()

TYPE	Notes	CBOR API
Unused7	A 7-bit value, set to zeros.	
Unused8	An 8-bit value, set to zeros.	
Unused16	A 16-bit value, set to zeros.	
Unused23	A 23-bit value, set to zeros.	
VENDOR_BIT	A 1 bit value, set to 1. It is placed in the highest bit position in the value.	
PHOENIX_BIT	A 1 bit value, set to 0. It is placed in the highest bit position in the value.	

Table 2: Phoenix Protocol types

3.4. StringRef

This protocol references strings in existing MIME objects by octet offset into the MIME object. This is called a StringRef. All strings can be referenced by using a total of 8 octets. The StringRef does not contain the string, it is a reference an existing string in a MIME object. A StringRef consists of two parts:

Name	Description	CBOR API
Offset	The octet count to the start of the string with zero being the first octet in the message.	cbor_Offset()
Length	The length in octets of the string.	cbor_Length()
StringRef	A String reference object.	cbor_StringRef()

Table 3: StringRef ABNF/CBOR Mapping

A StringRef over the wire is 8 octets in size.

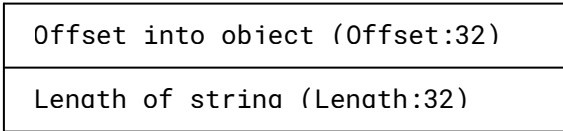


Figure 5: StringRef Format

ABNF:

4. Identifiers and Variables

Each item encoded and decoded in an application will be placed 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 onto the built in data types.

4.1. How to name a variable or identifier

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.

Or they can only be accessible within a 'namespace'. This is in the 'namespace' scope. See [Namespaces](#) (Section 9).

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:

```
identifier
: IDENTIFIER width?
;
```

Figure 6: identifier grammar

In the example [Figure 7](#), 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.
uint64_t TimeNow;        // Okay public, follows the rules.
float32_t Time9;         // Okay public, follows the rules.
uint64_t _Travel_Time    // Okay, 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.
  //
  void AMethodThatDoesSomething();
};
```

Figure 7: Sample with Internal 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:

```
internal_identifier : '_' identifier ;
```

Figure 8: internal_identifier grammar

Examples:

```
// Okay Internal, starts with an underscore.
// _NextSequenceNumber 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.

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

    // A public method that has access to all variables
    // including internal ones (_AName).
    // And only available inside the protocol implementation.
    //
    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 Collection object.
    //
    void _AnotherMethod();
};
```

Figure 9: Internal Scope Example

See [Section 12](#) 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 9](#) to understand namespaces and [Structs](#) ([Section 12](#)) 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 10: 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.
        //
        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.
        //
        void _AnotherMethod();
    };
};
```

Figure 11: Namespace Scope Example

See [Section 12](#) 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 and not a non-array variable.

Examples:

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

Figure 12: 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 13: Variable Fixed Size Array Examples

5.3. Variable Size Array '<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.

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 int8_t data elements.
//
float32_t FloatArray<3,16>;

// An string of 5 to 42 characters.
//
string StringVariable<5,42>;

// An string of 8 to unlimited characters.
//
string AString<8,*>;
```

Figure 14: 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 size zero to 17 in length.
//
string VariableName<17>;
```

Figure 15: Variable Size Array Max Size Examples

5.3.3. Variable Size Array With Default Size

<>

A shorthand for <0,*>.

Example:

```
uint64_t VariableName<>;
```

Figure 16: 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 17: Variable Array Examples

6. 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.

6.1. Integers

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

6.1.1. Unsigned Integer

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

`uintW_t`

An unsigned integer, 'W' bits wide. Where 'W' must be a multiple of 8.

ANTLR:

```
unsignedInteger
: 'uint8_t'
  | 'uint16_t'
  | 'uint32_t'
  | 'uint64_t'
  | bigNumUInt
;
```

Figure 18: unsignedInteger grammar

See [Section 6.1.1](#) for details.

Examples:

```
uint8_t EightBitsWide;
uint16_t SixteenBitsWide;
```

`uint8_t` An unsigned 8 bit integer. And the corresponding CBOR data type is unsigned int.

`uint16_t` An unsigned 16 bit integer. And the corresponding CBOR data type is unsigned int.

`uint32_t` An unsigned 32 bit integer. And the corresponding CBOR data type is unsigned int.

`uint64_t` An unsigned 64 bit integer. And the corresponding CBOR data type is unsigned int.

`uint72_t` An unsigned 72 bit integer. And the corresponding CBOR data type is bignum.

`uint128_t` An unsigned 128 bit integer. And the corresponding CBOR data type is bignum.

... `uintW_t` ... Assuming 'W' is more than 64 and a multiple of 8. An unsigned 'W'-bits integer. And the corresponding CBOR data type is bignum.

And the 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.

6.1.2. 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
: 'int8_t'
  | 'int16_t'
  | 'int32_t'
  | 'int64_t'
  | bigNumInt
;
```

Figure 19: *signedInteger* grammar

See [Section 6.1.1](#) for details.

Examples:

```
int8_t EightBitsWide;
int16_t SixteenBitsWide;
```

`int8_t` An signed 8 bit integer. And the corresponding CBOR data type is signed int.

`int16_t` An signed 16 bit integer. And the corresponding CBOR data type is signed int.

`int32_t` An signed 32 bit integer. And the corresponding CBOR data type is signed int.

`int64_t` An signed 64 bit integer. And the corresponding CBOR data type is signed int.

`int72_t` An signed 72 bit integer. And the corresponding CBOR data type is bignum.

`int128_t` An signed 128 bit integer. And the corresponding CBOR data type is bignum.

... `intLL_t` ... Assuming LL is more than 64 and a multiple of 8. An unsigned LL-bits integer. And the corresponding CBOR data type is bignum.

And the implementation transmits the signed integer value, not the data type. A `int128_t` unsigned integer with a value of one (1) would be transmitted as 0x01, a one octet value.

The 72, 128, and 'LL' 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.

6.1.3. Big Integers

CBOR supports signed and unsigned integers as 'bignum'. An implementation will transport non-standard size signed and unsigned integers as CBOR bignum.

6.2. Float

ANTLR:

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

Figure 20: float grammar

Examples:

```
float16_t Size;
float32_t BiggerSize;
```

Figure 21: 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.

6.3. 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 22: String Examples

ANTLR:

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

Figure 23: string grammar

6.4. 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 bitwise 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 is 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 of the using application design.

6.4.1. 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 24: 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;
```

6.4.2. 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 25: 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 \(Section 6.4.1\)](#)

```
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.

7. 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.

7.1. 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 26: Arrays Examples

7.2. 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,*>;
```

8. 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 27: declaration grammar

9. 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 transfer sets of data, each with a unique identifier and declare a type called UID.

In this simple example, it is conceivable 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.

Namespaces 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 uint64_t UID;

struct Packet
{
    UID Sequence;
    Blob ABlobOldData; // 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 uint64_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 uint64_t UID;

    struct Packet
    {
        UID Sequence;
        Blob ABlob0dData; // 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 uint64_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;
```

10. 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 ';' |  
| ('struct'|'class') identifier structBody ';' |  
| 'union' identifier unionBody ';' ;
```

Figure 28

In [Figure 29](#) and [Figure 30](#) 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 29

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 30

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

```
// Create a data type called 'PacketSequence'
//
typedef 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 31

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 demonstration only)
    //
    History.add(Cmd, Seq); // Error - should be (Seq, Cmd)

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

Figure 32

11. 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 simplify 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  
};
```

12. Structues

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

Struct ANTLR:

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

Figure 33

13. Union

todo

```
this is a test
```

14. Program

todo

```
this is a test
```

14.1. Versioning the Protocol

todo

15. IANA Considerations

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

16. Security Considerations

Robust digital certificate control. Especially with AUTHCERT_TLS.

MUCH MORE TODO ...

17. References

17.1. 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>>.

17.2. Informative References

- [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>>.

Appendix A. Complete ANGLR grammar

```

grammar cbor;

bigNumFloat : 'float' decimal '_t' ;

bigNumInt : 'int' decimal '_t' ;

bigNumUInt : 'uint' decimal '_t' ;

caseSpec
  : (('case' value':') | comment+ | passThrough+)
    (('case' value':') | comment+ | passThrough+)*
    declaration ';'
  ;

cborSpecification
  : specs+
  ;

comment : CommentOneLine
        | CommentMultiLine ;
CommentMultiLine : '/*' .*? '*/' ;
CommentOneLine : '//' ~[\r\n]+ ;

constantDef
  : 'const' typeSpecifier identifier '=' constant ';'
  ;

constant
  : decimal
    | float
    | hexadecimal
    | octal
  ;

dataType : typeSpecifier
  | typeSpecifier '[' value ']'
  | typeSpecifier '<' value? '>'
  | 'opaque' '[' value ']'
  | 'opaque' '<' value? '>'
  | 'string' '<' value? '>'
  | 'void'
  ;

decimal
  : DECIMAL width?
  ;

DECIMAL
  : ('-')? ([0-9])+
  ;

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

definition
  : typeDef
  | constantDef
  ;

enumBody
  : '{' (identifier '=' value) (',' identifier '=' value)* '}'

```

```

;
enumTypeSpec
: 'enum' enumBody
;

float
: 'float16_t'
| 'float32_t'
| 'float64_t'
| bigNumFloat
;
FLOAT
: ('-')? ([0-9])+ ('.' [0-9+]*)? (('e'|'E')('-')?[0-9+])?
;
floatValue
: FLOAT
;
hexadecimal
: HEXADECEIMAL width?
;
HEXADECEIMAL
: '0x' ([a-fA-F0-9])+
;
identifier
: IDENTIFIER width?
;
IDENTIFIER
: [a-zA-Z] ([a-zA-Z0-9_])*
;
ignoreTag : '[ignore]'
;
internal_identifier : '_' identifier ;
internalTag : '[internal]'
;
map
: 'map' '<' typeSpecifier ',' typeSpecifier '>'
;
method: dataType identifier '(' dataType identifier?
(',' dataType identifier?)* ')' ';'
;
multimap
: 'multimap' '<' typeSpecifier ',' typeSpecifier '>'
;
namespaceDef
: 'namespace' identifier ( ':' identifier )* ';'
;
namespace_identifier : '_' identifier ;
namespaceTag : '[namespace]'
;
octal : OCTAL width?
;
OCTAL
: '0' [1-7] ([0-7])*
;
overrideTag : '[override]'
;
PASS
: '%' ~[\n\r]+
| '%' [\n\r]+
;

```

```

        passThrough : PASS
        ;
        privateTag : '[private]'
        ;
procFirstArg
: 'void'
| dataType identifier?
;
program : 'program' identifier '{' version+ '}' '=' value ';'
;
publicTag : '[public]'
;
// A value enclosed in double quotes (")
QIDENTIFIERDOUBLE
: '"' [a-zA-Z] ([a-zA-Z0-9_])* '"'
;
QIDENTIFIERSINGLE
: '\'' [a-zA-Z] ([a-zA-Z0-9_])* '\''
;
specs
: structTypeSpec
| unionTypeSpec
| enumTypeSpec
| constantDef
| namespaceDef
| comment
| passThrough
| program
| declaration ';'
| typeDef
;
signedInteger
: 'int8_t'
| 'int16_t'
| 'int32_t'
| 'int64_t'
| bigNumInt
;
sortableTag : '[sortable]' ;
string : 'string' ;
structBody
: '{'
    ((declaration ';' ) | comment+ | method+ | passThrough+)
    ((declaration ';' ) | comment+ | method+ | passThrough+)*
    '}'
;
structTypeSpec
: ('struct' | 'class') structBody
;
tags : ignoreTag
| internalTag
| overrideTag
| privateTag
| namespaceTag
| sortableTag
;
typeDef
: 'typedef' declaration ';'
| 'enum' identifier enumBody ';'

```

```

    | ('struct'|'class') identifier structBody ';'
    | 'union' identifier unionBody ';'
    ;
typeSpecifier
: 'bool'
| enumTypeSpec
| float
| identifier
| map
| multimap
| 'opaque'
| signedInteger
| 'string'
| structTypeSpec
| unionTypeSpec
| unsignedInteger
;
unionBody
: 'switch' '(' declaration ')' '{'
  caseSpec
  caseSpec*
  ('default' ':' declaration ';')?
  '}'
;
unionTypeSpec
: 'union' unionBody
; unsignedInteger
: 'uint8_t'
  | 'uint16_t'
  | 'uint32_t'
  | 'uint64_t'
  | bigNumUInt
  ;
  value
: constant
| identifier
| QIDENTIFIERSINGLE
| QIDENTIFIERDOUBLE
;
version : 'version' identifier '{' versionMethod+ '}' '=' value ';'
;
  versionMethod: dataType identifier
  '(' ((dataType identifier?
  (',' dataType identifier?)* ) | 'void') ')' '=' value ';'
;
  width
  : (':' DECIMAL)
  ;
  WS
: [ \t\r\n]+ -> skip
;

```

Figure 34: Complete ANGLR grammar

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](tel:1+208-806-1358)

Email: DouglasRoyer@gmail.com

URI: <https://DougRoyer.US>