

CBOR Maintenance and Extensions
Internet-Draft
Intended status: Standards Track
Expires: 2 September 2026

C. Bormann
Universität Bremen TZI
B. Moran
Arm Limited
1 March 2026

CDDL Module Structure
draft-ietf-cbor-cddl-modules-06

Abstract

At the time of writing, the Concise Data Definition Language (CDDL) is defined by RFC 8610 and RFC 9682 as well as RFC 9165 and RFC 9741. The latter two have used the extension point provided in RFC 8610, the `_control` operator.

As CDDL is being used in larger projects, the need for features has become known that cannot be easily mapped into this single extension point.

The present document defines a backward- and forward-compatible way to add a module structure to CDDL.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at <https://cbor-wg.github.io/cddl-modules/draft-ietf-cbor-cddl-modules.html>. Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-ietf-cbor-cddl-modules/>.

Discussion of this document takes place on the CBOR Maintenance and Extensions Working Group mailing list (<mailto:cbor@ietf.org>), which is archived at <https://mailarchive.ietf.org/arch/browse/cbor/>. Subscribe at <https://www.ietf.org/mailman/listinfo/cbor/>.

Source for this draft and an issue tracker can be found at <https://github.com/cbor-wg/cddl-modules>.

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 2 September 2026.

Copyright Notice

Copyright (c) 2026 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. Introduction	3
1.1. Conventions and Definitions	3
2. Module superstructure	3
2.1. Compatibility	4
2.2. Namespacing	4
2.3. "Directives", the "module"	5
2.4. Naming and Finding Modules	5
2.5. Basic Set of Directives	6
2.6. Explicit selection of names	7
include	7
import	8
2.7. Tool Support for Command-Line Control	9
3. Security Considerations	10
4. IANA Considerations	11
5. References	11
5.1. Normative References	11
5.2. Informative References	11
Appendix A. ABNF Specification	12
Appendix B. A CDDL Tool that Implements CDDL Module Structure .	13
Acknowledgments	13

Authors' Addresses	13
------------------------------	----

1. Introduction

At the time of writing, the Concise Data Definition Language (CDDL) is defined by RFC 8610 and RFC 9682 as well as RFC 9165 and RFC 9741. The latter two have used the extension point provided in RFC 8610, the `_control operator_`.

As CDDL is being used in larger projects, the need for features has become known that cannot be easily mapped into this single extension point.

The present document defines a backward- and forward-compatible way to add a module structure to CDDL.

The functionality of the present document is considered a core part of what has colloquially been called CDDL 2.0, and is intended to be used with (and orthogonal to) other recent specifications such as a small set of CDDL grammar updates [RFC9682] and an additional exercise of the control operator extension point [RFC9741] (a follow-up to [RFC9165]).

1.1. Conventions and Definitions

The Terminology from [RFC8610] applies.

Some examples use CDDL definitions from [RFC9052].

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 [BCP14] (RFC2119) (RFC8174) when, and only when, they appear in all capitals, as shown here.

2. Module superstructure

`_Compatibility_`: `bidirectional` (both backward and forward)

Originally, CDDL was used for small data models that could be expressed in a few lines. As the size of data models that need to be expressed in CDDL has increased, the need to modularize and re-use components is increasing.

CDDL as documented in [RFC8610] (`_basic CDDL_`) has been designed with a crude form of composition: Concatenating a number of CDDL snippets creates a valid CDDL data model unless there is a name collision (`_identical_` redefinition is allowed to facilitate this approach). With larger models, managing the overall name space to avoid collisions becomes more pressing.

In CDDL's original composition model, the knowledge which CDDL snippets need to be concatenated in order to obtain the desired data model lives entirely outside the CDDL snippets. With the module structure defined in the present document, rule sets are packaged as modules and referenced from other modules, providing methods for control of namespace pollution.

Further work may be expended on unambiguous referencing into evolving specifications ("versioning") and selection of alternatives (as was emulated with snippets in Section 11 of [RFC8428]. Note that one approach for expressing variants is demonstrated in [useful] based on Section 4 of [RFC9165]). Potential further work is outlined in Sections 4 and A.2 of [I-D.bormann-cbor-cddl-2-draft].

2.1. Compatibility

To achieve the module structure in a way that is friendly to existing environments that operate with CDDL and basic CDDL implementations, we add a super-syntax (similar to the way pragmas are often added to a language), by carrying them in `_directives_`. Directives are parsed as comments in basic CDDL, so that each module source file can at the same time be basic CDDL on its own. This enables forward compatibility: a single file can be designed to serve both as valid basic CDDL and as a module with directives that specify how additional rule definitions are to be imported from other source files.

This bidirectional compatibility is useful to allow CDDL model designers to start using directives before the bulk of the tools that process CDDL has been updated to implement the present specification.

2.2. Namespacing

When importing rules from other modules, there is the potential for name collisions. This is exacerbated when the modules evolve, which may lead to the introduction of a name into an imported module that is also used (likely in a different way) in the importing module.

To be able to manage names in such a way that collisions can be avoided, we introduce means to prepend a prefix to the names of rules being imported: the "as" clause.

2.3. "Directives", the "module"

This specification introduces `_directives_` into CDDL. A single CDDL file becomes a `_module_` by processing the (zero or more) directives in it.

The semantics of the module are independent of the module(s) using it, however, importing a module may involve transforming its rule names into a new namespace (Section 2.2).

Directives are parsed as comments in basic CDDL, so they do not interfere with forward compatibility.

In the CDDL module structure, lines that start with the prefix `;` are parsed as directives.

2.4. Naming and Finding Modules

We assume that module names are filenames taken from one of several source directories available to the CDDL module structure processor via the environment. This avoids the need to nail down brittle pathnames or (partial?) URIs into the CDDL files.

The exact way how these source directories and possibly a precedence between them are established is intentionally not fully defined in this specification; it is expected that this will be specified in the context of the models just as the way they are intended to be used will be. (A more formal structure may follow later.)

In the module structure implementation that is part of the CDDL Tool described in Appendix B, the set of sources is determined from an environment variable, `CDDL_INCLUDE_PATH`, which is modeled after usual command-line search paths. It is a colon-separated list of pathnames to directories, with one special feature: an empty element points to the tool's own collection. This collection contains fragments of extracted CDDL from published RFCs, using names such as `rfc9052`.

(Future versions might augment this with Web extractors and/or ways to extract CDDL modules from sites such as github and from Internet-Drafts; see Appendix A.2 of [I-D.bormann-cbor-cddl-2-draft] for some design considerations.)

The default `CDDL_INCLUDE_PATH` is:

`.:`

That is: files are found in the current directory (`.`) and, if not found there, `cddltool` collection.

In the examples that follow, a `cddl` command line will be shown (starting with an isolated `$` sign as in a shell command) with the module-structured CDDL input; the resulting basic CDDL will be shown separately.

2.5. Basic Set of Directives

Two groups of directives are defined at this point:

- * `include`, which includes all the rules from a module (which includes the ones imported/included there, transitively), or specific explicitly selected rules (clause ending in "from");
- * `import`, which is similar to `include` but includes only those rules from the module that are referenced from the importing module, implicitly or explicitly (clause ending in "from"), including the rules that are referenced from these rules, transitively.

The `include` function is more useful for composing a single model from parts controlled by one author, while the `import` function is more about treating a module as a library:

The way an `import` works is shown by this simple example:

```
$ cddl -2tcddl -
start = COSE_Key
;# import rfc9052
```

This results in the following CDDL 1.0 specification:

```
start = COSE_Key
COSE_Key = {
  1 => tstr / int,
  ? 2 => bstr,
  ? 3 => tstr / int,
  ? 4 => [+ tstr / int],
  ? 5 => bstr,
  * label => values,
}
label = int / tstr
values = any
```

This is appropriate for using libraries that are well known to the importing specification. However, if it is not acceptable that the library can pollute the namespace of the importing module, the `import` directive can specify a namespace prefix ("as" clause):

```
$ cddlrc -2tcddl -
start = cose.COSE_Key
;# import rfc9052 as cose
```

This results in the following CDDL 1.0 specification:

```
start = cose.COSE_Key
cose.COSE_Key = {
  1 => tstr / int,
  ? 2 => bstr,
  ? 3 => tstr / int,
  ? 4 => [+ tstr / int],
  ? 5 => bstr,
  * cose.label => cose.values,
}
cose.label = int / tstr
cose.values = any
```

Note how the imported names are prefixed with `cose.` as specified in the import directive, but CDDL prelude (Appendix D of [RFC8610]) names such as `tstr` and `any` are not.

2.6. Explicit selection of names

Both import and include directives can be augmented by an explicit mentioning of rule names (clause ending in "from") or a wild-card "*" for all rules.

include

Starting with include:

```
$ cddlrc -2tcddl -
mydata = { * label => values }
;# include label, values from rfc9052
```

With include, only exactly the rules mentioned are included:

```
mydata = { * label => values }
label = int / tstr
values = any
```

The module from which rules are explicitly imported can be namespaced:

```
$ cddlrc -2tcddl -
mydata = { * label => values }
;# include cose.label, cose.values from rfc9052 as cose
```

Again, only exactly the rules mentioned are included:

```
mydata = { * label => values }
cose.label = int / tstr
cose.values = any
```

import

Both examples would work exactly the same with import, as the included rules do not reference anything else from the included module.

An import however also draws in the transitive closure of the rules referenced:

```
$ cddl -2tcddl -
mydata = {Fritz: cose.empty_or_serialized_map}
;# import cose.empty_or_serialized_map from rfc9052 as cose
```

The transitive closure of the rules mentioned is included:

===== NOTE: '\ ' line wrapping per RFC 8792 =====

```
mydata = {Fritz: cose.empty_or_serialized_map}
cose.empty_or_serialized_map = bstr .cbor cose.header_map / bstr .\
                                                                    size 0
cose.header_map = {
  cose.Generic_Headers,
  * cose.label => cose.values,
}
cose.Generic_Headers = (
  ? 1 => int / tstr,
  ? 2 => [+ cose.label],
  ? 3 => tstr / int,
  ? 4 => bstr,
  ? (5 => bstr // 6 => bstr),
)
cose.label = int / tstr
cose.values = any
```

The import statement can also request an alias for an imported name:

```
$ cddl -2tcddl -
mydata = {Fritz: cose.empty_or_serialized_map}
;# import empty_or_serialized_map from rfc9052 as cose
```

Note how an additional rule provides an alias for empty_or_serialized_map that does not have the namespace prefix:

===== NOTE: '\ ' line wrapping per RFC 8792 =====

```
mydata = {Fritz: cose.empty_or_serialized_map}
empty_or_serialized_map = cose.empty_or_serialized_map
cose.empty_or_serialized_map = bstr .cbor cose.header_map / bstr .\
                                                                    size 0

cose.header_map = {
  cose.Generic_Headers,
  * cose.label => cose.values,
}
cose.Generic_Headers = (
  ? 1 => int / tstr,
  ? 2 => [+ cose.label],
  ? 3 => tstr / int,
  ? 4 => bstr,
  ? (5 => bstr // 6 => bstr),
)
cose.label = int / tstr
cose.values = any
```

2.7. Tool Support for Command-Line Control

Tools may provide a convenient way to initiate the processing of directives from the command line.

A tool may provide a way to specify a root for the module tree from the command line:

```
$ cddl -2tcddl -icose=rfc9052 -scose.COSE_Key
```

The command line argument `-icose=rfc9052` is a shortcut for

```
import rfc9052 as cose
```

Together with the start (root) rule name, `cose.COSE_Key`, supplied by `-scose.COSE_Key`, this results in the following CDDL 1.0 specification:

```
$.start.$ = cose.COSE_Key
cose.COSE_Key = {
  1 => tstr / int,
  ? 2 => bstr,
  ? 3 => tstr / int,
  ? 4 => [+ tstr / int],
  ? 5 => bstr,
  * cose.label => cose.values,
}
cose.label = int / tstr
cose.values = any
```

In other words, the module synthesized from the command line had an empty CDDL file, which therefore was not provided (no 襲 was given on the command line).

3. Security Considerations

The module structure specified in this document is not believed to create additional security considerations beyond the general security considerations in Section 5 of [RFC8610].

Implementations that employ the module structure defined in this document need to ascertain the provenance of the modules they combine into the CDDL models they employ operationally. This specification does not define how the source directories accessed via the `CDDL_INCLUDE_PATH` are populated; this process needs to undergo the same care and scrutiny as any other introduction of source code into a build environment; the possibility of supply-chain attacks on the modules imported needs to be considered.

Specifically, implementations that rely on model-based input validation for enforcing certain properties of the data structure ingested (which, if not validated, could lead to malfunctions such as crashes and remote code execution) need to be particularly careful about the data models they apply, including their provenance and potential changes of these properties that upgrades to the referenced modules may (inadvertently or as part of an attack) cause. More generally speaking, implementations should strive to be robust against limitations of the model-based input validation mechanisms and their implementations that they employ.

In applications that dynamically acquire models and dereference module references in these, the security considerations of dereferenceable identifiers apply (see [I-D.bormann-t2trg-deref-id] for a more extensive discussion of dereferenceable identifiers).

4. IANA Considerations

This document has no IANA actions.

5. References

5.1. Normative References

- [BCP14] Best Current Practice 14,
<<https://www.rfc-editor.org/info/bcp14>>.
At the time of writing, this BCP comprises the following:
- 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>>.
- 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>>.
- [RFC7405] Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, DOI 10.17487/RFC7405, December 2014, <<https://www.rfc-editor.org/rfc/rfc7405>>.
- [RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/rfc/rfc8610>>.
- [STD68] Internet Standard 68,
<<https://www.rfc-editor.org/info/std68>>.
At the time of writing, this STD comprises the following:
- Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <<https://www.rfc-editor.org/info/rfc5234>>.

5.2. Informative References

- [cddl-c] "CDDL conversion utilities", n.d., <<https://github.com/cabo/cddl-c>>.
- [I-D.bormann-cbor-cddl-2-draft] Bormann, C., "CDDL 2.0 and beyond -- a draft plan", Work in Progress, Internet-Draft, draft-bormann-cbor-cddl-2-

draft-07, 30 August 2025,
<<https://datatracker.ietf.org/doc/html/draft-bormann-cbor-cddl-2-draft-07>>.

[I-D.bormann-t2trg-deref-id]

Bormann, C. and C. Ams^端ss, "The "dereferenceable identifier" pattern", Work in Progress, Internet-Draft, draft-bormann-t2trg-deref-id-07, 24 February 2026, <<https://datatracker.ietf.org/doc/html/draft-bormann-t2trg-deref-id-07>>.

[RFC8428] Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C. Bormann, "Sensor Measurement Lists (SenML)", RFC 8428, DOI 10.17487/RFC8428, August 2018, <<https://www.rfc-editor.org/rfc/rfc8428>>.

[RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", STD 96, RFC 9052, DOI 10.17487/RFC9052, August 2022, <<https://www.rfc-editor.org/rfc/rfc9052>>.

[RFC9165] Bormann, C., "Additional Control Operators for the Concise Data Definition Language (CDDL)", RFC 9165, DOI 10.17487/RFC9165, December 2021, <<https://www.rfc-editor.org/rfc/rfc9165>>.

[RFC9682] Bormann, C., "Updates to the Concise Data Definition Language (CDDL) Grammar", RFC 9682, DOI 10.17487/RFC9682, November 2024, <<https://www.rfc-editor.org/rfc/rfc9682>>.

[RFC9741] Bormann, C., "Concise Data Definition Language (CDDL): Additional Control Operators for the Conversion and Processing of Text", RFC 9741, DOI 10.17487/RFC9741, March 2025, <<https://www.rfc-editor.org/rfc/rfc9741>>.

[useful] "Useful CDDL", n.d., <<https://github.com/cbor-wg/cddl/wiki/Useful-CDDL>>.

Appendix A. ABNF Specification

This section specifies the grammar employed by the module structure directives using the ABNF language defined in [STD68] and [RFC7405].

```
directive = ";"# RS (%s"import" / %s"include") RS [from-clause]
           filename [as-clause] CRLF
from-clause = 1*(id-or-all [","] RS) %s"from" RS
as-clause = RS %s"as" RS id
filename = 1*("-" / "." / %x30-39 / %x41-5a / "_" / %x61-7a)
id = (" $" / %x40-5a / "_" / %x61-7a)
    *(" $" / %x30-39 / %x40-5a / "_" / %x61-7a)
id-or-all = id / "*"
RS = 1*WS
WS = SP
SP = %x20
CRLF = %x0A / %x0D.0A
```

Appendix B. A CDDL Tool that Implements CDDL Module Structure

This appendix is for information only.

A CDDL conversion tool is available [cddlcl] that can process module-structured CDDL models into basic CDDL models; the latter can then be processed by other CDDL tools such as the one described in Appendix F of [RFC8610].

A typical command line involving both tools mentioned might be:

```
cddlcl -2 -tcddl mytestfile.cddl | cddl - gp 10
```

The CDDL conversion tool can be installed on a system with a modern Ruby (Ruby version 3.0) via:

```
gem install cddlcl
```

The present document assumes the use of cddlcl of at least version 0.2.5.

Acknowledgments

TODO acknowledge.

Authors' Addresses

Carsten Bormann
Universit t Bremen TZI
Postfach 330440
D-28359 Bremen
Germany
Phone: +49-421-218-63921
Email: cabo@tzi.org

Brendan Moran
Arm Limited
Email: brendan.moran.ietf@gmail.com