Thing-to-Thing Research Group
Internet-Draft
Intended status: Experimental
Expires: March 15, 2018

K. Hartke
Universitaet Bremen TZI
September 11, 2017

The Constrained RESTful Application Language (CoRAL)
draft-hartke-t2trg-coral-03

Abstract

The Constrained RESTful Application Language (CoRAL) defines a data model and interaction model as well as two specialized serialization formats for the description of typed connections between resources on the Web ("links"), possible operations on such resources ("forms"), and simple resource metadata.

Discussion Venue

This Internet-Draft is discussed on the Thing-to-Thing Research Group (T2TRG) mailing list <https://www.irtf.org/mailman/listinfo/t2trg>.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Constrained RESTful Application Language (CoRAL) is a language for the description of typed connections between resources on the Web ("links"), possible operations on such resources ("forms"), as well as simple resource metadata.

CoRAL is intended for driving automated software agents that navigate a Web application based on a standardized vocabulary of link and form relation types. CoRAL is designed to be used in conjunction with a Web transfer protocol such as the Hypertext Transfer Protocol (HTTP) [RFC7230] or the Constrained Application Protocol (CoAP) [RFC7252].

This document defines the CoRAL data and interaction model as well as two specialized CoRAL serialization formats.

The CoRAL data and interaction model is a superset of the Web linking model described in RFC 5988 [I-D.nottingham-rfc5988bis]. The data model consists of two elements: _links_, which describe relationships between pairs of resources and the type of those relationships, and _forms_, which describe possible operations on resources and the type of those operations. Additionally, the model can describe simple resource metadata in a style similar to the Resource Description Framework (RDF) [W3C.REC-rdf11-concepts-20140225]. In contrast to RDF, the focus of CoRAL however is on the interaction with resources, not just on the relationships between them. The interaction model

derives from HTML 5 [W3C.REC-html51-20161101] and specifies how an automated software agent can navigate between resources by following links and perform operations on resources by submitting forms.

The primary CoRAL serialization format is a compact, binary encoding of links and forms in Concise Binary Object Representation (CBOR) [RFC7049]. It is intended for environments with constraints on power, memory, and processing resources [RFC7228] and shares many similarities with the message format of the Constrained Application Protocol (CoAP) [RFC7252]: It uses numeric identifiers instead of verbose strings for link and form relation types and pre-parses URIs into (what CoAP considers to be) their components, which simplifies URI processing greatly. As a result, link serializations are often much more compact than equivalent serializations in CoRE Link Format [RFC6690], including its CBOR variant [I-D.ietf-core-links-json]. Additionally, the format supports the serialization of forms, which the CoRE Link Format does not support.

The second serialization format is a light, textual encoding of links and forms that is intended to be easy to read and write by humans. The format is used for giving examples throughout the document and is stylistically similar to Turtle [W3C.REC-turtle-20140225], although it is not a syntax for RDF.

1.1. Requirements Notation

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. Examples

2.1. Web Linking

At its core, CoRAL is just yet another serialization format for Web links. For example, if an HTTP client sends the following request:

```
GET /TheBook/chapter3 HTTP/1.1
Host: example.com
```

and receives the following response:

```
HTTP/1.1 200 OK
Content-Type: text/coral

#using <http://www.iana.org/assignments/relation/>

next    <./chapter4>
icon    </favicon.png>
license <http://creativecommons.org/licenses/by/4.0/>
```

then the representation contains the following three links:

- one link of type "http://www.iana.org/assignments/relation/next" from <http://example.com/TheBook/chapter3> to <http://example.com/TheBook/chapter4>,

- one link of type "http://www.iana.org/assignments/relation/icon" from <http://example.com/TheBook/chapter3> to <http://example.com/favicon.png>, and

- one link of type "http://www.iana.org/assignments/relation/license" from <http://example.com/TheBook/chapter3> to <http://creativecommons.org/licenses/by/4.0/>.

This representation is equivalent to the following Link header field [I-D.nottingham-rfc5988bis]:

```
Link: <./chapter4>; rel="next",
      </favicon.png>; rel="icon",
      <http://creativecommons.org/licenses/by/4.0/>; rel="license"
```

and the following HTML 5 [W3C.REC-html51-20161101] link elements:

```
<link rel="next" href="./chapter4">
<link rel="icon" href="/favicon.png">
<link rel="license"
      href="http://creativecommons.org/licenses/by/4.0/">
```

2.2. Links, Forms, and Metadata

In its entirety, CoRAL is an expressive language for describing Web links between resources, possible operations on these resources, and simple resource metadata. For example, if an HTTP client sends the following request:

```
GET /tasks HTTP/1.1
Host: example.com
```

and receives the following response:

```
HTTP/1.1 200 OK
Content-Type: text/coral

#using <http://example.org/vocabulary#>
#using coral = <urn:ietf:rfc:XXXX#>

task </tasks/1> {
   description "Pick up the kids"
}

task </tasks/2> {
   description "Return the books to the library"
   coral:delete -> DELETE </tasks/2>
}

coral:create -> POST </tasks> [coral:accept "example/task"]
```

then the representation contains the following six elements:

- o one link of type "http://example.org/vocabulary#task" from <http://example.com/tasks> to <http://example.com/tasks/1>,

- o one link of type "http://example.org/vocabulary#description" from <http://example.com/tasks/1> to "Pick up the kids",

- o one link of type "http://example.org/vocabulary#task" from <http://example.com/tasks> to <http://example.com/tasks/2>,

- o one link of type "http://example.org/vocabulary#description" from <http://example.com/tasks/2> to "Return the books to the library",

- o one form of type "urn:ietf:rfc:XXXX#delete" that can be used to delete <http://example.com/tasks/2> by making a DELETE request to <http://example.com/tasks/2>, and

- o one form of type "urn:ietf:rfc:XXXX#create" that can be used to create a new item in <http://example.com/tasks> by making a POST request to <http://example.com/tasks> with an "example/task" payload.

3. Data and Interaction Model

The Constrained RESTful Application Language (CoRAL) is designed for building Web-based applications [W3C.REC-webarch-20041215] in which automated software agents navigate between resources by following links and perform operations on resources by submitting forms.

3.1. Browsing Context

Borrowing from HTML 5 [W3C.REC-html51-20161101], each such agent maintains a _browsing context_ in which the representations of Web resources are processed. (In HTML 5, the browsing context typically corresponds to a tab or window in a Web browser.)

A browsing context has a _session history_, which lists the resource representations that the agent has processed, is processing, or will process. At any time, one representation in each browsing context is designated the _active_ representation.

A session history consists of a flat list of session history entries. Each _session history entry_ consists of a resource representation and the Internationalized Resource Identifier (IRI) [RFC3987] that was used to retrieve the representation. An entry may additionally have other information associated with it. New entries are added to the session history as the agent navigates from resource to resource.

3.2. Documents

A resource representation in one of the CoRAL serialization formats is called a CoRAL _document_. The IRI that was used to retrieve a document is called the document's _retrieval context_.

A CoRAL document consists of a list of zero or more links and forms, collectively called _elements_. CoRAL serialization formats may define additional types of elements for efficiency or convenience, such as a base IRI for relative IRI references.

3.3. Links

A _link_ describes a relationship between two resources on the Web. It consists of a _link context_, a _link relation type_, and a _link target_. A link may additionally have a nested list of zero or more elements, which take the place of link target attributes in CoRAL.

A link can be viewed as a statement of the form "_link context_ has a _link relation type_ resource at _link target_", where the link target may be further described by nested links and forms.

The link relation type identifies the semantics of a link. In HTML 5 and RFC 5988 Link headers, a link relation type is typically denoted by a registered name, such as "stylesheet" or "icon". In CoRAL, a link relation type is denoted by an IRI or an unsigned integer. IRIs allow the creation of new, unique relation types in a decentralized fashion, but impose some overhead on the size of representations. Unsigned integers on the other hand minimize the overhead of link

relation types in constrained environments, but necessitate the assignment of values by a registry.

The link context and the link target are both resources. Resources in CoRAL are denoted either by an IRI reference or a literal, similar to RDF. If the scheme of the IRI indicates a Web transfer protocol such as HTTP or CoAP, then an agent can dereference the IRI and navigate their browsing context to the referenced resource; this is called _following the link_. A literal directly identifies a value. CoRAL supports Boolean values, integers, floating-point values, byte strings, and text strings as literals.

A link can occur as a top-level element in a document or as a nested element within a link. When a link occurs as a top-level element in a document, the link context is implicitly the document's retrieval context. When a link occurs within a link, the link context of the inner link is the link target of the outer link.

There are no restrictions on the cardinality of links; there can be multiple links to and from a particular target, and multiple links of the same or different types between a given link context and target. However, the CoRAL data model constrains the description of a web of resources to a graph in tree shape: Links between linked resources can only be described by further nesting links.

3.4. Forms

A _form_ provides instructions to an agent for performing an operation on a Web resource. It consists of a _form context_, a _form relation type_, a _request method_, and a _submission IRI_. Additionally, a form MAY be accompanied by _form data_.

A form can be viewed as an instruction of the form "To perform a _form relation type_ operation on _form context_, make a _request method_ request to _submission IRI_", where the payload of the request may be further described by form data.

The form relation type identifies the semantics of the operation. Like link relation types, the form relation type can be denoted by an IRI or an unsigned integer. Link and form relation types constitute different namespaces, though.

The form context is the resource on which an operation is ultimately performed. To perform the operation, an agent must construct a request with the specified request method and submission IRI. The set of possible request methods is defined by the protocol identified by the scheme of the submission IRI. The submission IRI typically

refers to the form context, but MAY refer to a different resource. Constructing and sending the request is called _submitting the form_.

If a form is accompanied by form data (see Section 3.5 below), the agent MUST also construct and include a payload in the request that matches the specifications of the form data when submitting the form.

A form can occur as a top-level element in a document or as a nested element within a link. When a form occurs as a top-level element in a document, the form context is implicitly the document's retrieval context. When a form occurs within a link, the form context is the link target of the enclosing link.

3.5. Form Data

Form data provides instructions for agents to construct a request payload. It consists of a list of zero or more _form fields_. Each form field consists of a _form field name_ and a _form field value_.

Form fields can either directly identify data items that need to be included in the request payload or reference an external resource (such as a schema) that describes the data. Additionally, they can provide other information, such as acceptable serialization formats.

The form field name identifies the semantics of the form field. Like link and form relation types, a form field name is denoted by an IRI or an unsigned integer.

The form field value can be an IRI, a Boolean value, an integer, a floating-point value, a byte string, or a text string.

3.6. Navigation

An agent begins interacting with an application through a GET request on an _entry point IRI_. The entry point IRI is the only IRI an agent is expected to know before interacting with an application. From there, the agent is expected to make all requests by following links and submitting forms that are provided by the server in responses. The entry point IRI can be obtained by manual configuration or some discovery process.

If dereferencing the entry point IRI yields a CoRAL document or any other representation that implements the CoRAL data and interaction model, then the agent proceeds as follows:

1. The first step for the agent is to decide what to do next, i.e., which type of link to follow or form to submit, based on the link relation types and form relation types it understands.

2. The agent finds the link(s) or form(s) with the given relation type in the active representation. This may yield one or more candidates, from which the agent must select the most appropriate one in the next step. The set of candidates MAY be empty, for example, if a transition is not supported or allowed.

3. The agent selects one of the candidates based on the metadata associated with the link(s) or form(s). Metadata typically includes the media type of the target resource representation, the IRI scheme, the request method, and other information that describe the element.

4. The agent resolves the IRI reference in the link or form to obtain the _request IRI_. Fragment identifiers are not part of the request IRI and MUST be separated from the rest of the IRI prior to a dereference. The request IRI may need to be converted to a URI (see Section 3.1 of RFC 3987 [RFC3987]) for protocols that do not support IRIs.

5. The agent constructs a new request with the request IRI. If the agent follows a link, the request method MUST be GET. If the agent submits a form, the request method MUST be the one specified in the form. The agent SHOULD set HTTP header fields and CoAP request options according to provided metadata (e.g., set the HTTP Accept header field or the CoAP Accept option when the media type of the target resource is provided). In the case of a form with form data, the agent MUST include a request payload that matches the specifications of the form data.

6. The agent sends the request and retrieves the response.

7. If a fragment identifier was separated from the request IRI, the agent dereferences the fragment identifier within the retrieved representation.

8. The agent _updates the session history_: It removes all the entries in the browsing context's session history after the current entry. Then it appends a new entry at the end of the history representing the new resource.

9. Finally, if response contains a CoRAL document, the agent can again decide what to do next.

3.7. History Traversal

An agent can additionally navigate a browsing context by traversing the browsing context's session history. An agent can _traverse the

session history_ by updating the active representation to the that entry.

4. Binary Format

This section defines the encoding of documents in the CoRAL binary format.

A document in the binary format is a data item in Concise Binary Object Representation (CBOR) [RFC7049]. The structure of this data item is presented in the Concise Data Definition Language (CDDL) [I-D.ietf-cbor-cddl].

4.1. Data Structure

The data structure of a document in the binary format is made up of three kinds of elements: links and forms, as defined by the CoRAL data model, and base IRI directives. Base IRI directives provide a way to encode IRI references with a common base more efficiently.

Elements are processed in the order they appear in the document. Document processors need to maintain an _environment_ while iterating an array of elements. The environment consists of three variables: a _current context IRI_, a _current base IRI_, and a _current relation type_. The current context IRI and current base IRI are initially both set to the document's retrieval context. The current relation type is initially set to the unsigned integer zero.

4.1.1. Documents

The body of a document in the binary format is encoded as an array of zero or more links, forms, and directives.

```
body = [*(link / form / directive)]
```

4.1.2. Links

A link is encoded as an array that consists of the unsigned integer 2, followed by the link relation type and the link target, optionally followed by a link body that contains nested elements.

```
link = [link: 2, relation, target, ?body]
```

The link relation type is encoded either as a text string containing an absolute IRI reference or as an (unsigned or negative) integer representing the difference to the current relation type. A link is processed by updating the current relation type to the result of

adding the specified integer (or zero in the case of a text string) to the current relation type.

```
relation = text / int
```

The link target is denoted by an IRI reference or represented by a literal value. The IRI reference may be relative or absolute and is resolved against the current base IRI. The encoding of IRI references in the binary format is described in Appendix C. The link target MAY be null, which indicates that the link target is an unidentified resource.

```
target = iri / literal / null

literal = bool / int / float / bytes / text
```

The array of elements in the link body (if any) MUST be processed in a fresh environment. The current context IRI and current base IRI in the new environment are initially both set to the link target of the enclosing link. The current relation type in the new environment is initially set to the current relation type.

4.1.3. Forms

A form is encoded as an array that consists of the unsigned integer 3, followed by the form relation type, the submission method, and a submission IRI reference, optionally followed by form data.

```
form = [form: 3, relation, method, iri, ?form-data]
```

The form relation type is encoded and processed in the same way as a link relation type; see Section 4.1.2.

The method is encoded as a text string or an unsigned integer that refers to one of the request methods defined by the protocol that is identified by the scheme of the submission IRI. Method identifiers in the form of text strings are constrained to the range of US-ASCII characters and are case-insensitive.

```
method = text / uint
```

For HTTP [RFC7231], the method MUST be encoded as a text string; the set of possible values is maintained in the IANA HTTP Method Registry. For CoAP [RFC7252], the method MUST be encoded as an unsigned integer (e.g., the unsigned integer 2 for the POST method); the set of possible values is maintained in the IANA CoAP Method Codes Registry.

The submission IRI reference may be relative or absolute and is resolved against the current base IRI. The encoding of IRI references in the binary format is described in Appendix C.

4.1.3.1. Form Data

Form data is encoded as an array of zero or more name-value pairs.

```
form-data = [*(form-field-name, form-field-value)]
```

Form data (if any) MUST be processed in a fresh environment. The current context IRI and current base IRI in the new environment are initially both set to the submission IRI of the enclosing form. The current relation type in the new environment is initially set to the current relation type.

A form field name is encoded and processed in the same way as a link relation type; see Section 4.1.2.

```
form-field-name = text / uint
```

A form field value can be an IRI reference, Boolean value, integer, floating-point value, byte string, or text string. An IRI reference may be relative or absolute and is resolved against the current base IRI. The encoding of IRI references in the binary format is described in Appendix C.

```
form-field-value = iri / bool / int / float / bytes / text
```

4.1.3.2. Short Forms

Forms in certain shapes can be encoded in a more efficient manner using short forms. The following short forms are available:

```
form =/ [form.create: 4, ?accept: uint .size 2]

form =/ [form.update: 5]

form =/ [form.delete: 6]
```

If the scheme of the submission IRI indicates HTTP, the short forms expand as follows:

```
[4]     ->  [3, "urn:ietf:rfc:XXXX#create", "POST", []]
[4, x]  ->  [3, "urn:ietf:rfc:XXXX#create", "POST", [],
               ["urn:ietf:rfc:XXXX#accept", x]]
[5]     ->  [3, "urn:ietf:rfc:XXXX#update", "PUT", []]
[6]     ->  [3, "urn:ietf:rfc:XXXX#delete", "DELETE", []]
```

If the scheme of the submission IRI indicates CoAP, the short forms expand as follows:

```
[4]     ->  [3, "urn:ietf:rfc:XXXX#create", 2, []]
[4, x]  ->  [3, "urn:ietf:rfc:XXXX#create", 2, [],
               ["urn:ietf:rfc:XXXX#accept", x]]
[5]     ->  [3, "urn:ietf:rfc:XXXX#update", 3, []]
[6]     ->  [3, "urn:ietf:rfc:XXXX#delete", 4, []]
```

The form relation types and form field names in the above expansions are defined in Appendix A.

4.1.4. Directives

Directives provide the ability to manipulate the environment when processing a list of elements. There is one directive available: the Base URI directive.

```
directive = base-directive
```

4.1.4.1. Base URI Directives

A Base IRI directive is encoded as an array that consists of the unsigned integer 1, followed by an IRI reference.

```
base-directive = [base: 1, iri]
```

The encoding of IRI references in the binary format is described in Appendix C.

The directive is processed by resolving the IRI reference against the current context IRI and assigning the result to the current base IRI.

5. Textual Format

This section defines the syntax of documents in the CoRAL textual format using two grammars: The lexical grammar defines how Unicode characters are combined to form line terminators, white space, comments, and tokens. The syntactic grammar defines how the tokens are combined to form documents. Both grammars are presented in Augmented Backus-Naur Form (ABNF) [RFC5234].

A document in the textual format is a Unicode string in a Unicode encoding form [UNICODE]. The media type for such documents is "text/coral". The "charset" parameter is not used; charset information is transported inside the document in the form of an OPTIONAL Byte Order Mark (BOM). The use of the UTF-8 encoding scheme [RFC3629], without a BOM, is RECOMMENDED.

5.1. Lexical Structure

The lexical structure of a document in the textual format is made up of four basic elements: line terminators, white space, comments, and tokens. Of these, only tokens are significant in the syntactic grammar. There are four kinds of tokens: identifiers, IRI references, literals, and punctuators.

When several lexical grammar rules match a sequence of characters in a document, the longest match takes priority.

5.1.1. Line Terminators

Line terminators divide text into lines. A line terminator is any Unicode character with Line_Break class BK, CR, LF, or NL. However, any CR character that immediately precedes a LF character is ignored. (This affects only the numbering of lines in error messages.)

5.1.2. White Space

White space is a sequence of one or more white space characters. A white space character is any Unicode character with the White_Space property.

5.1.3. Comments

Comments are sequences of characters that are ignored when parsing text into tokens. Single-line comments begin with the characters "//" and extend to the end of the line. Delimited comments begin with the characters "/*" and end with the characters "*/". Delimited comments can occupy a portion of a line, a single line, or multiple lines.

Comments do not nest. The character sequences "/*" and "*/" have no special meaning within a single-line comment, and the character sequences "//" and "/*" have no special meaning within a delimited comment.

5.1.4. Identifiers

Identifier tokens are user-defined symbolic names. The rules for identifiers correspond exactly to those recommended by the Unicode Standard Annex #31 [UNICODE-UAX31]: An identifier consists of a Unicode character in the ID_Start class followed by zero or more Unicode characters in the ID_Continue class.

```
identifier = ID_Start *ID_Continue
```

```
ID_Start = <Any character in the ID_Start class>

ID_Continue = <Any character in the ID_Continue class>
```

All identifiers MUST be converted into Unicode Normalization Form C (NFC), as defined by the Unicode Standard Annex #15 [UNICODE-UAX15]. Comparison of identifiers is based on NFC and is case-sensitive (unless otherwise noted).

5.1.5. IRI References

An IRI reference is a Unicode string that conforms to the syntax defined in RFC 3987 [RFC3987]. An IRI reference may be absolute or relative and may contain a fragment identifier. IRI references are enclosed in angle brackets ("<" and ">").

```
iri = %x3C IRI-reference %x3E

IRI-reference = <Defined in RFC 3987>
```

5.1.6. Literals

A literal is a textual representation of a value. There are six types of literals: Boolean, integer, floating-point, byte string, text string, and null.

5.1.6.1. Boolean Literals

The case-sensitive literals "true" and "false" denote the Boolean values true and false, respectively.

```
boolean = %x74.72.75.65 / %x66.61.6C.73.65
```

5.1.6.2. Integer Literals

Integer literals denote integer values of unspecified precision. By default, integer literals are expressed in decimal, but they can also be specified in an alternate base using a prefix. Binary literals begin with "0b", octal literals begin with "0o", and hexadecimal literals begin with "0x".

Decimal literals contain the digits "0" through "9". Binary literals contain "0" and "1", octal literals contain "0" through "7", and hexadecimal literals contain "0" through "9" as well as "A" through "F" in upper- or lowercase.

Negative integers are expressed by prepending a minus sign ("-").

```
integer = [%x2D] (decimal / binary / octal / hexadecimal)

decimal = 1*DIGIT

binary = %x30 (%x42 / %x62) 1*BINDIG

octal = %x30 (%x4F / %x6F) 1*OCTDIG

hexadecimal = %x30 (%x58 / %x78) 1*HEXDIG

DIGIT = %x30-39

BINDIG = %x30-31

OCTDIG = %x30-37

HEXDIG = %x30-39 / %x41-46 / %x61-66
```

5.1.6.3. Floating-point Literals

Floating-point literals denote floating-point values of unspecified precision.

Floating-point literals consist of a sequence of decimal digits followed by either a fraction, an exponent, or both. The fraction consists of a decimal point (".") followed by a sequence of decimal digits. The exponent consists of the upper- or lowercase letter "e" followed by an optional sign and a sequence of decimal digits that indicate a power of 10 by which the value preceding the "e" is multiplied.

Negative floating-point values are expressed by prepending a minus sign ("-").

```
floating-point = [%x2D] 1*DIGIT [fraction] [exponent]

fraction = %x2E 1*DIGIT

exponent = (%x45 / %x65) [%x2B / %x2D] 1*DIGIT
```

Floating-point literals can additionally denote the special "Not-a-Number" (NaN) value, positive infinity, and negative infinity. The NaN value is produced by the case-sensitive token "NaN". The two infinite values are produced by the case-sensitive tokens "+Infinity" (or simply "Infinity") and "-Infinity".

```
floating-point =/ %x4E.61.4E
```

```
floating-point =/ [%x2B] %x49.6E.66.69.6E.69.74.79

floating-point =/ %x2D.49.6E.66.69.6E.69.74.79
```

5.1.6.4. Byte String Literals

A byte string literal consists of a prefix and zero or more bytes encoded in Base16, Base32, or Base64 [RFC4648] and enclosed in single quotes. Byte string literals encoded in Base16 begin with "h" or "b16", byte string literals encoded in Base32 begin with "b32", and byte string literals encoded in Base64 begin with "b64".

```
bytes = base16 / base32 / base64

base16 = (%x68 / %x62.31.36) SQUOTE <Base16 encoded data> SQUOTE

base32 = %x62.33.32 SQUOTE <Base32 encoded data> SQUOTE

base64 = %x62.36.34 SQUOTE <Base64 encoded data> SQUOTE

SQUOTE = %x27
```

5.1.6.5. Text String Literals

A text string literal consists of zero or more Unicode characters enclosed in double quotes. It can include simple escape sequences (such as \t for the tab character) as well as hexadecimal and Unicode escape sequences.

```
text = DQUOTE *(char / %x5C escape) DQUOTE

char = <Any character except %x22, %x5C, and line terminators>

escape = simple-escape / hexadecimal-escape / unicode-escape

simple-escape = %x30 / %x62 / %x74 / %x6E / %x76

simple-escape =/ %x66 / %x72 / %x22 / %x27 / %x5C

hexadecimal-escape = (%x78 / %x58) 2HEXDIG

unicode-escape = %x75 4HEXDIG / %x55 8HEXDIG

DQUOTE = %x22
```

An escape sequence denotes a single Unicode code point. For hexadecimal and Unicode escape sequences, the code point is expressed by the hexadecimal number following the "\x", "\X", "\u", or "\U"

prefix. Simple escape sequences indicate the code points listed in Table 1.

Escape Sequence	Character Name	Code Point
\0	Null	U+0000
\b	Backspace	U+0008
\t	Character Tabulation	U+0009
\n	Line Feed	U+000A
\v	Line Tabulation	U+000B
\f	Form Feed	U+000C
\r	Carriage Return	U+000D
\"	Quotation Mark	U+0022
\'	Apostrophe	U+0027
\\	Reverse Solidus	U+005C

Table 1: Simple Escape Sequences

5.1.6.6. Null Literal

The literals "null" and "?" denote the intentional absence of any value.

```
null = %x6E.75.6C.6C / %x3F
```

5.1.7. Punctuators

Punctuator tokens are used for grouping and separating.

```
punctuator = "#" | ":" | "[" | "]" | "{" | "}" | "=" | "->"
```

5.2. Syntactic Structure

The syntactic structure of a document in the textual format is made up of three kinds of elements: links and forms, as defined by the CoRAL data model, and directives. Directives provide a way to make documents easier to read and write by defining a base IRI for relative IRI references and introducing shorthands for link and form relation types.

Elements are processed in the order they appear in the document. Document processors need to maintain an _environment_ while iterating a list of elements. The environment consists of three variables: a _current context IRI_, a _current base IRI_, and a mapping from identifiers to IRIs. The current context IRI and current base IRI

are initially both set to the document's retrieval context. The mapping from identifiers to IRIs is initially empty.

5.2.1. Documents

The body of a document in the textual format consists of zero or more links, forms, and directives.

```
body = *(link / form / directive)
```

5.2.2. Links

A link consists of the link relation type, followed by the link target, optionally followed by a link body enclosed in curly brackets ("{" and "}").

```
link = relation target ["{" body "}"]
```

The link relation type is denoted either by an absolute IRI reference, a simple name, a qualified name, or an unsigned integer.

```
relation = iri / simple-name / qualified-name / integer
```

An absolute IRI reference MUST be resolved and normalized according to the IRI scheme. (It doesn't matter what base IRI it is resolved against, since it is always an absolute reference.)

A simple name consists of an identifier. It is resolved to an IRI by looking up the empty string in the mapping from identifiers to IRIs in the current environment and concatenating the looked up IRI with the specified identifier. It is an error if the empty string is not present in the mapping.

```
simple-name = identifier
```

A qualified name consists of two identifiers separated by a colon (":"). It is resolved to an IRI by looking up the identifier on the left hand side in the mapping from identifiers to IRIs in the current environment and concatenating the looked up IRI with the identifier on the right hand side. It is an error if the identifier on the left hand side is not present in the mapping.

```
qualified-name = identifier ":" identifier
```

The link target is denoted by an IRI reference or represented by a value literal. The IRI reference may be relative or absolute and is resolved against the current base IRI. If the link target is null, the link target is an unidentified resource.

```
target = iri / literal / null

literal = boolean / integer / floating-point / bytes / text
```

The list of elements in the link body (if any) MUST be processed in a fresh environment. The current context IRI and current base IRI in this environment are initially both set to the link target of the enclosing link. The mapping from identifiers to IRIs is initially set to a copy of the mapping from identifiers to IRIs in the current environment.

5.2.3. Forms

A form consists of the form relation type, followed by a "->" token, a method identifier, and a submission IRI reference, optionally followed by form data enclosed in square brackets ("[" and "]").

```
form = relation "->" method iri ["[" form-data "]"]
```

The form relation type is denoted in the same way as a link relation type; see Section 5.2.2.

The method identifier refers to one of the request methods defined by the protocol that is identified by the scheme of the submission IRI. Method identifiers are constrained to the range of US-ASCII characters and are case-insensitive.

```
method = identifier
```

For HTTP [RFC7231], the set of possible method identifiers is maintained in the IANA HTTP Method Registry. For CoAP [RFC7252], the set of possible method identifiers is maintained in the IANA CoAP Method Codes Registry.

The submission IRI reference may be relative or absolute and is resolved against the current base IRI.

5.2.3.1. Form Data

Form data consists of zero or more space-separated name-value pairs.

```
form-data = *(form-field-name form-field-value)
```

Form data MUST be processed in a fresh environment. The current context IRI and current base IRI in this environment are initially both set to the submission IRI of the enclosing form. The mapping from identifiers to IRIs is initially set to a copy of the mapping from identifiers to IRIs in the current environment.

The form field name is denoted in the same way as a link relation type; see Section 5.2.2.

```
form-field-name = iri / simple-name / qualified-name / integer
```

The form field value can be an IRI reference, Boolean literal, integer literal, floating-point literal, byte string literal, or text string literal. An IRI reference may be relative or absolute and is resolved against the current base IRI.

```
form-field-value = iri / boolean / integer

form-field-value =/ floating-point / bytes / text
```

5.2.4. Directives

Directives provide the ability to manipulate the environment when processing a list of elements. All directives start with a number sign ("#") followed by a directive identifier. Directive identifiers are constrained to the range of US-ASCII characters and are case-insensitive.

The following directives are available: Base IRI directives and Using directives.

```
directive = base-directive / using-directive
```

5.2.4.1. Base IRI Directives

A Base IRI directive consists of a number sign ("#"), followed by the case-insensitive identifier "base", followed by an IRI reference.

```
base-directive = "#" "base" iri
```

The directive is processed by resolving the IRI reference against the current context IRI and assigning the result to the current base IRI.

5.2.4.2. Using Directives

A Using directive consists of a number sign ("#"), followed by the case-insensitive identifier "using", optionally followed by an identifier and an equals sign ("="), followed by an absolute IRI reference. If the identifier is not specified, it is assumed to be the empty string.

```
using-directive = "#" "using" [identifier "="] iri
```

The directive is processed by resolving the IRI reference to an IRI, normalizing the IRI according to the IRIs scheme, and adding the specified identifier and resulting IRI to the mapping from identifiers to IRIs in the current environment. (It doesn't matter what the IRI reference is resolved against, since it is always an absolute reference.) It is an error if the identifier is already present in the mapping.

6. Usage Considerations

This section discusses some considerations in creating CoRAL-based applications and managing link and form relation types.

6.1. Specifying CoRAL-based Applications

CoRAL-based applications implement the Web architecture [W3C.REC-webarch-20041215] and are centered around orthogonal specifications for identification, interaction, and representation:

- o Resources are identified by IRIs or represented by value literals.

- o Interactions are based on the hypermedia interaction model of the Web and the methods provided by the Web transfer protocol. The semantics of possible interactions are identified by link and form relation types.

- o Representations are CoRAL documents encoded in the binary format defined in Section 4 or the textual format defined in Section 5. Depending on the application, additional representation formats may be used.

Specifications for CoRAL-based applications need to specify the specific protocols and vocabulary used in the application. This SHOULD include at least the following items:

- o IRI schemes identifying the Web transfer protocol(s) used by the application.

- o Internet media types that identify the representation format(s) used by the application, including the media type(s) of the CoRAL serialization format(s).

- o Link relation types that identify the semantics of links.

- o Form relation types that identify the semantics of forms. Additionally, for each form relation type, the permissible request method(s).

o Form field names that identify the semantics of form fields.

6.1.1. Naming Resources

Resource names [RFC3986] [RFC3987] are a cornerstone of Web-based applications. They enable uniform identification of resources and are used every time a client interacts with a server or a resource representation needs to refer to another resource.

URIs and IRIs often include structured application data in the path and query components, such as paths in a filesystem or keys in a database. It is a common practice in many HTTP-based applications to make this part of the documented application interface, i.e., to prescribe a fixed URI template that is hard-coded in implementations. However, there are a number of problems with this practice [RFC7320].

In CoRAL-based applications, resource names are not part of the application interface; they are an implementation detail. The specification of a CoRAL-based application MUST NOT mandate any particular form of resource name structure. RFC 7320 [RFC7320] describes the problematic practice of fixed URI structures in more detail and provides some acceptable alternatives.

6.2. Minting New Relation Types

New link relation types, form relation types, and form field names can be minted by defining an IRI [RFC3987] that uniquely identifies the item. Although the IRI can point to a resource that contains a definition of the semantics of the relation type, clients SHOULD NOT automatically access that resource to avoid overburdening its server. The IRI SHOULD be under the control of the person or party defining it, or be delegated to them.

Link relation types registered in the IANA Link Relations Registry, such as collection" [RFC6573] or "icon" [W3C.REC-html51-20161101], can be used in CoRAL by prepending the IRI <http://www.iana.org/assignments/relation/> to the registered name:

```
#using iana = <http://www.iana.org/assignments/relation/>

iana:collection </items>
iana:icon       </favicon.png>
```

A good source for link relation types for resource metadata are RDF predicates [W3C.REC-rdf11-concepts-20140225]. An RDF statement says that some relationship, indicated by a predicate, holds between two resources. RDF predicates and link relation types can therefore

often be used interchangeably. For example, a CoRAL document could describe its maker using the FOAF vocabulary [FOAF] as follows:

```
#using iana = <http://www.iana.org/assignments/relation/>
#using foaf = <http://xmlns.com/foaf/0.1/>

foaf:maker ? {
   iana:type       <http://xmlns.com/foaf/0.1/Person>
   foaf:familyName "Hartke"
   foaf:givenName  "Klaus"
   foaf:mbox       <mailto:hartke@tzi.org>
}
```

6.3. Registering Relation Types

IRIs that identify link relation types, form relation types, and form field names do not have to be registered. The use of DNS names in them allows for decentralized creation of new IRIs without the risk of collisions. However, IRIs can be relatively verbose and can impose a high overhead in CoRAL representations. This can be a problem in constrained environments [RFC7228].

CoRAL alternatively allows the use of unsigned integers to identify link relation types, form relation types, and form field names. These impose a much smaller overhead, but need to be assigned by a registry to avoid collisions.

This document does not create a registry for such integers. Instead, a "profile" parameter [RFC6906] is specified for use with the media types for CoRAL documents in the binary and textual format. The parameter specifies which registry to use.

The registries are identified for this purpose by a URI [RFC3986]. For example, a CoRAL document that uses the registry identified by the URI <http://example.com/coral> uses the following media type:

```
application/coral+cbor; profile="http://example.com/coral"
```

The URI serves only as an identifier; it does not necessarily have to be dereferencable (or even use a dereferencable URI scheme). It is permissible, though, to use a dereferencable URI and serve a representation that provides information about the registry in a human- or machine-readable way. (The format of such a representation is outside the scope of this document.)

For simplicity, a CoRAL document can only use at most one registry at a time. The "profile" parameter of the CoRAL media types MUST contain a single URI, not a whitespace-separated list of URIs as

recommended in RFC 6906 [RFC6906]. If the "profile" parameter is absent, the profile specified in Appendix B is assumed.

A registry SHOULD map each unsigned integer to the full IRI that identifies the link relation type, form relation type, or form field name. The namespaces for these three kinds of identifiers are disjoint, i.e., the same integer may be assigned to a link relation type, form relation type, and form field name without ambiguity. Once an integer has been assigned, the assignment MUST NOT be changed or removed. A registry MAY provide additional information about an assignment (for example, whether a link relation type is deprecated).

In CoAP, media types (including specific values for their parameters) are encoded as a small, unsigned integer, called the content format. For use with CoAP, each CoRAL registry needs to register a new content format in the IANA CoAP Content-Formats Registry. Each such registered content format MUST specify a CoRAL media type with a "profile" parameter containing the registry URI.

6.4. Expressing Target Attributes

Link target attributes defined for use with RFC 6690 (e.g., "type", "hreflang", "media", "ct", "rt", "if", "sz") can be expressed in CoRAL by nesting links under the respective link and specifying the attribute name appended to the IRI <http://www.example.org/TBD/> as the link relation type. The target of such nested links MUST be a text string literal:

```
#using iana = <http://www.iana.org/assignments/relation/>
#using attr = <http://www.example.org/TBD/>

iana:item </patches/1> {
   attr:type "application/json-patch+json"
   attr:ct   "51"
}
```

[[NOTE TO RFC EDITOR: Please replace all occurrences of "http://www.example.org/TBD/" with a RFC-Editor-controlled IRI.]]

Link target attributes that actually describe the link and not the link target (e.g., "rel", "anchor", "rev", "title", and "title*") are excluded from this provision and MUST NOT occur in a CoRAL document.

6.5. Embedding CoRAL in CBOR Structures

Data items in the CoRAL binary format (Section 4) can be embedded in other CBOR [RFC7049] data structures. Specifications using CDDL

[I-D.ietf-cbor-cddl] SHOULD reference the following CDDL definitions for this purpose:

```
CoRAL-Body = body

CoRAL-Link = link

CoRAL-Form = form

CoRAL-IRI = iri
```

7. Security Considerations

Parsers of CoRAL documents must operate on input that is assumed to be untrusted. This means that parsers MUST fail gracefully in the face of malicious inputs. Additionally, parsers MUST be prepared to deal with resource exhaustion that results, for example, from the allocation of big data items or exhaustion of the stack depth (stack overflow). See Section 8 of RFC 7049 [RFC7049] for security considerations relating to parsing CBOR in particular.

Implementers of the CoRAL textual format need to consider the security aspects of handling Unicode input. See the Unicode Standard Annex #36 [UNICODE-UAX36] for security considerations relating to visual spoofing and misuse of character encodings. See Section 10 of RFC 3629 [RFC3629] for security considerations relating to UTF-8.

CoRAL makes extensive use of IRIs and URIs. See Section 8 of RFC 3987 [RFC3987] for security considerations relating to IRIs and Section 7 of RFC 3986 [RFC3986] for security considerations relating to URIs.

CoRAL is intended to be used in conjunction with a Web transfer protocol such as HTTP or CoAP. See Section 9 of RFC 7320 [RFC7230], Section of 9 of RFC 7231 [RFC7231], etc. for security considerations relating to HTTP. See Section 11 of RFC 7252 [RFC7252] for security considerations relating to CoAP.

CoRAL does not define any specific mechanisms for the confidentiality and integrity of CoRAL documents. It relies on application layer or transport layer security mechanisms, such as Transport Layer Security (TLS) [RFC5246].

CoRAL documents and the structure of a web of resources revealed from automatically following links can disclose personal information and other sensitive information. Implementations need to prevent the unintentional disclosure of such information. See Section of 9 of RFC 7231 [RFC7231] for additional considerations.

The security of applications using CoRAL can depend on the proper preparation and comparison of internationalized strings. For example, such strings can be used to make authentication and authorization decisions, and the security of an application could be compromised if an entity providing a given string is connected to the wrong account or online resource based on different interpretations of the string. See RFC 6943 [RFC6943] for security considerations relating to identifiers in IRIs and other locations.

Applications using CoRAL ought to consider the attack vectors opened by automatically following, trusting, or otherwise using links and forms in CoRAL documents. In particular, a server that is authoritative for the CoRAL representation of a resource may not necessarily be the authoritative source for nested links and forms.

8. IANA Considerations

8.1. Media Type "application/coral+cbor"

This document registers the media type "application/coral+cbor" according to the procedures of BCP 13 [RFC6838].

Type name:
   application

Subtype name:
   coral+cbor

Required parameters:
   N/A

Optional parameters:
   N/A

Encoding considerations:
   binary

Security considerations:
   See Section 7 of [I-D.hartke-t2trg-coral].

Interoperability considerations:
   N/A

Published specification:
   [I-D.hartke-t2trg-coral]

Applications that use this media type:
   See Section 1 of [I-D.hartke-t2trg-coral].

Fragment identifier considerations:
   As specified for "application/cbor".

Additional information:
   Deprecated alias names for this type: N/A
   Magic number(s): N/A
   File extension(s): N/A
   Macintosh file type code(s): N/A

Person & email address to contact for further information:
   See the Author's Address section of [I-D.hartke-t2trg-coral].

Intended usage:
   COMMON

Restrictions on usage:
   N/A

Author:
   See the Author's Address section of [I-D.hartke-t2trg-coral].

Change controller:
   IESG

Provisional registration?
   No

8.2. Media Type "text/coral"

This document registers the media type "text/coral" according to the procedures of BCP 13 [RFC6838] and guidelines in RFC 6657 [RFC6657].

Type name:
   text

Subtype name:
   coral

Required parameters:
   N/A

Optional parameters:
   N/A

Encoding considerations:
   See Section 5 of [I-D.hartke-t2trg-coral].

Security considerations:

See Section 7 of [I-D.hartke-t2trg-coral].

Interoperability considerations:
N/A

Published specification:
[I-D.hartke-t2trg-coral]

Applications that use this media type:
See Section 1 of [I-D.hartke-t2trg-coral].

Fragment identifier considerations:
N/A

Additional information:
Deprecated alias names for this type: N/A
Magic number(s): N/A
File extension(s): .coral
Macintosh file type code(s): TEXT

Person & email address to contact for further information:
See the Author's Address section of [I-D.hartke-t2trg-coral].

Intended usage:
COMMON

Restrictions on usage:
N/A

Author:
See the Author's Address section of [I-D.hartke-t2trg-coral].

Change controller:
IESG

Provisional registration?
No

8.3. CoAP Content Formats

This document registers CoAP content formats for the media types "application/coral+cbor" and "text/coral" according to the procedures of BCP 13 [RFC6838].

- Media Type: application/coral+cbor
  Content Coding: identity
  ID: TBD (maybe 63)
  Reference: [I-D.hartke-t2trg-coral]

o Media Type: text/coral
  Content Coding: identity
  ID: TBD (maybe 10063)
  Reference: [I-D.hartke-t2trg-coral]

9. References

9.1. Normative References

[I-D.ietf-cbor-cddl]
Birkholz, H., Vigano, C., and C. Bormann, "Concise data definition language (CDDL): a notational convention to express CBOR data structures", draft-ietf-cbor-cddl-00 (work in progress), July 2017.

[I-D.nottingham-rfc5988bis]
Nottingham, M., "Web Linking", draft-nottingham-rfc5988bis-08 (work in progress), August 2017.

[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>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC3987] Duerst, M. and M. Suignard, "Internationalized Resource Identifiers (IRIs)", RFC 3987, DOI 10.17487/RFC3987, January 2005, <https://www.rfc-editor.org/info/rfc3987>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] 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>.

[RFC6657] Melnikov, A. and J. Reschke, "Update to MIME regarding "charset" Parameter Handling in Textual Media Types", RFC 6657, DOI 10.17487/RFC6657, July 2012, <https://www.rfc-editor.org/info/rfc6657>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC6943] Thaler, D., Ed., "Issues in Identifier Comparison for Security Purposes", RFC 6943, DOI 10.17487/RFC6943, May 2013, <https://www.rfc-editor.org/info/rfc6943>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[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>.

[UNICODE] The Unicode Consortium, "The Unicode Standard", <http://www.unicode.org/versions/latest/>.

Note that this reference is to the latest version of Unicode, rather than to a specific release. It is not expected that future changes in the Unicode specification will have any impact on this document.

[UNICODE-UAX15]
The Unicode Consortium, "Unicode Standard Annex #15: Unicode Normalization Forms", <http://unicode.org/reports/tr15/>.

[UNICODE-UAX31]
The Unicode Consortium, "Unicode Standard Annex #31: Unicode Identifier and Pattern Syntax", <http://unicode.org/reports/tr31/>.

[UNICODE-UAX36]
The Unicode Consortium, "Unicode Standard Annex #36: Unicode Security Considerations", <http://unicode.org/reports/tr36/>.

9.2. Informative References

[FOAF] Brickley, D. and L. Miller, "FOAF Vocabulary Specification 0.99", January 2014, <http://xmlns.com/foaf/spec/20140114.html>.

[I-D.ietf-core-links-json] Li, K., Rahman, A., and C. Bormann, "Representing Constrained RESTful Environments (CoRE) Link Format in JSON and CBOR", draft-ietf-core-links-json-09 (work in progress), July 2017.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, DOI 10.17487/RFC5789, March 2010, <https://www.rfc-editor.org/info/rfc5789>.

[RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, DOI 10.17487/RFC5952, August 2010, <https://www.rfc-editor.org/info/rfc5952>.

[RFC6573] Amundsen, M., "The Item and Collection Link Relations", RFC 6573, DOI 10.17487/RFC6573, April 2012, <https://www.rfc-editor.org/info/rfc6573>.

[RFC6690] Shelby, Z., "Constrained RESTful Environments (CoRE) Link Format", RFC 6690, DOI 10.17487/RFC6690, August 2012, <https://www.rfc-editor.org/info/rfc6690>.

[RFC6903] Snell, J., "Additional Link Relation Types", RFC 6903, DOI 10.17487/RFC6903, March 2013, <https://www.rfc-editor.org/info/rfc6903>.

[RFC6906] Wilde, E., "The 'profile' Link Relation Type", RFC 6906, DOI 10.17487/RFC6906, March 2013, <https://www.rfc-editor.org/info/rfc6906>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <https://www.rfc-editor.org/info/rfc7228>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/info/rfc7252>.

[RFC7320] Nottingham, M., "URI Design and Ownership", BCP 190, RFC 7320, DOI 10.17487/RFC7320, July 2014, <https://www.rfc-editor.org/info/rfc7320>.

[RFC8132] van der Stok, P., Bormann, C., and A. Sehgal, "PATCH and FETCH Methods for the Constrained Application Protocol (CoAP)", RFC 8132, DOI 10.17487/RFC8132, April 2017, <https://www.rfc-editor.org/info/rfc8132>.

[W3C.REC-html51-20161101]
Faulkner, S., Eicholz, A., Leithead, T., and A. Danilo, "HTML 5.1", World Wide Web Consortium Recommendation REC-html51-20161101, November 2016, <https://www.w3.org/TR/2016/REC-html51-20161101>.

[W3C.REC-rdf11-concepts-20140225]
Cyganiak, R., Wood, D., and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax", World Wide Web Consortium Recommendation REC-rdf11-concepts-20140225, February 2014, <http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225>.

[W3C.REC-turtle-20140225]
Prud'hommeaux, E. and G. Carothers, "RDF 1.1 Turtle", World Wide Web Consortium Recommendation REC-turtle-20140225, February 2014, <http://www.w3.org/TR/2014/REC-turtle-20140225>.

[W3C.REC-webarch-20041215]
Jacobs, I. and N. Walsh, "Architecture of the World Wide Web, Volume One", World Wide Web Consortium Recommendation REC-webarch-20041215, December 2004, <http://www.w3.org/TR/2004/REC-webarch-20041215>.

Appendix A. Core Vocabulary

This section defines the core vocabulary for CoRAL. It is RECOMMENDED that all profiles assign an unsigned integer to each of these link relation types, form relation types, and form field names.

[[NOTE TO RFC EDITOR: Please replace all occurrences of "urn:ietf:rfc:XXXX#" with a RFC-Editor-controlled IRI.]]

A.1. Link Relation Types

<http://www.iana.org/assignments/relation/type>
: Indicates that the link's context is an instance of the type specified as the link's target; see Section 6 of RFC 6903 [RFC6903].

  This link relation type serves in CoRAL the same purpose as the RDF predicate identified by the IRI <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>.

<http://www.iana.org/assignments/relation/item>
: Indicates that the link's context is a collection and that the link's target is a member of that collection; see Section 2.1 of RFC 6573 [RFC6573].

<http://www.iana.org/assignments/relation/collection>
: Indicates that the link's target is a collection and that the link's context is a member of that collection; see Section 2.2 of RFC 6573 [RFC6573].

A.2. Form Relation Types

<urn:ietf:rfc:XXXX#create>
: Indicates that the form's context is a collection and that a new item can be created in that collection by submitting the form with a representation. This form relation type is typically used with the POST method [RFC7231] [RFC7252].

<urn:ietf:rfc:XXXX#update>
: Indicates that the form's context can be updated by submitting a representation. This form relation type is typically used with the PUT method [RFC7231] [RFC7252] or PATCH method [RFC5789] [RFC8132].

<urn:ietf:rfc:XXXX#delete>
: Indicates that the form's context can be deleted. This form relation type is typically used with the DELETE method [RFC7231] [RFC7252].

<urn:ietf:rfc:XXXX#search>
Indicates that the form's context can be searched by submitting a search query. This form relation type is typically used with the POST method [RFC7231] [RFC7252] or FETCH method [RFC8132].

A.3. Form Field Names

<urn:ietf:rfc:XXXX#accept>
Specifies an acceptable content type or content format for the request payload. There MAY be multiple form fields with this name. If a form does not include a form field with this name, the server accepts any or no request payload, depending on the form relation type.

For HTTP, the content type MUST be specified as a text string in the format specified in Section 3.1.1.1 of RFC 7231 [RFC7231]; the set of possible values is maintained in the IANA Media Types Registry.

For CoAP, the content format MUST be specified as an unsigned integer; the set of possible values is maintained in the IANA CoAP Content-Formats Registry.

Appendix B. Default Profile

This section defines a default registry that is assumed when a CoRAL media type without a "profile" parameter is used.

Link Relation Types

0 = <http://www.iana.org/assignments/relation/type>
1 = <http://www.iana.org/assignments/relation/item>
2 = <http://www.iana.org/assignments/relation/collection>

Form Relation Types

0 = <urn:ietf:rfc:XXXX#create>
1 = <urn:ietf:rfc:XXXX#update>
2 = <urn:ietf:rfc:XXXX#delete>
3 = <urn:ietf:rfc:XXXX#search>

Form Fields

0 = <urn:ietf:rfc:XXXX#accept>

Appendix C. CBOR-encoded IRI References

URI references [RFC3986] and, secondarily, IRI references [RFC3987] and are the most common usage of resource identifiers in hypertext representation formats such as HTML 5 [W3C.REC-html51-20161101] and the CoRE Link Format [RFC6690]. They encode the components of a resource identifier either as an absolute URI/IRI or as a relative reference that is resolved against a base URI/IRI.

URI and IRI references are sequences of characters chosen from limited subsets of the repertoires of US-ASCII and Unicode characters, respectively. The individual components of a URI or IRI are delimited by several reserved characters, which necessitates the use of percent-encoding for reserved characters in a non-delimiting function. The resolution of references involves parsing URI/IRI references into their components, combining the components with those of the base URI/IRI, merging paths, removing dot segments, and recomposing the result into a URI/IRI reference string.

Overall, proper URI processing can be quite complex, which can be a problem in particular in constrained environments [RFC7228] with severe code size limitations. As a result, many implementations in such environments choose to implement only an ad-hoc, informally-specified, bug-ridden, non-interoperable subset of half of RFC 3986 (or less).

This section specifies CBOR-encoded IRI References, a serialization format for IRI references that encodes their components as CBOR data items rather than text. Given a CBOR implementation, typical operations on IRI references such as parsing, reference resolution and comparison can be implemented much more easily than with the text-based format. A full implementation that covers all corner cases of the specification can be implemented in a relatively small amount of code.

CBOR-encoded IRI References are not capable of expressing all IRI references permitted by RFC 3987 [RFC3987]. The supported subset includes all CoAP URIs [RFC7252] and most HTTP URIs [RFC7230].

C.1. Data Structure

The encoding is very similar to the encoding of the request URI in CoAP messages [RFC7252]. The components of an IRI reference are encoded as a sequence of _options_. Each option consists of an _option number_ identifying the type of option (IRI scheme, host name, etc.) and the _option value_.

```
iri = [?(scheme:    1, text),
       ?(host.name: 2, text //
         host.ip:   3, bytes .size 4 / bytes .size 16),
       ?(port:      4, uint .size 2),
       ?(path.type: 5, path-type),
       *(path:      6, text),
       *(query:     7, text),
       ?(fragment:  8, text)]

path-type = &(absolute-path:   0,
              append-path:     1,
              relative-path:   2,
              append-relation: 3)
```

C.2. Options

The following options are defined:

scheme
Specifies the IRI scheme. The option value MUST match the "scheme" rule defined in Section 3.1 of RFC 3986.

host.name
Specifies the host of the IRI authority as a registered name.

host.ip
Specifies the host of the IRI authority as an IPv4 address (4 bytes) or an IPv6 address (16 bytes).

port
Specifies the port number. The option value MUST be an unsigned integer in the range 0 to 65535 (inclusive).

path.type
Specifies the type of the IRI path for reference resolution. Possible values are 0 (absolute-path), 1 (append-path), 2 (relative-path), and 3 (append-relation).

path
Specifies one segment of the IRI path. This option can occur more than once.

query
Specifies one argument of the IRI query. This option can occur more than once.

fragment
Specifies the fragment identifier.

The value of the "host.name", "path", "query", and "fragment" options can be any Unicode string. No percent-encoding is performed.

C.3. Properties

A sequence of options is considered _well-formed_ if:

- the sequence of options is empty or starts with a "scheme", "host.name", "host.ip", "port", "path.type", "path", "query", or "fragment" option;

- a "scheme" option is followed by either a "host.name" or "host.ip" option;

- a "host.name" option is followed by a "port" option;

- a "host.ip" option is followed by a "port" option;

- a "port" option is followed by a "path", "query", or "fragment" option or is at the end of the sequence;

- a "path.type" option is followed by a "path", "query", or "fragment" option or is at the end of the sequence;

- a "path" option is followed by a "path", "query", or "fragment" option or is at the end of the sequence;

- a "query" option is followed by a "query" or "fragment" option or is at the end of the sequence; and

- a "fragment" option is at the end of the sequence.

A well-formed sequence of options is considered _absolute_ if the sequence of options starts with a "scheme" option. A well-formed sequence of options is considered _relative_ if the sequence of options is empty or starts with an option other than the "scheme" option.

An absolute sequence of options is considered _normalized_ if the result of resolving the sequence of options against any base IRI reference is equal to the input. (It doesn't matter what it is resolved against, since it is already absolute.)

C.4. Reference Resolution

This section defines how to resolve a CBOR-encoded IRI reference that might be relative to a given base IRI.

Applications MUST resolve a well-formed sequence of options `href` against an absolute sequence of options `base` by using an algorithm that is functionally equivalent to the following Python 3.5 code.

<CODE BEGINS>

```
def resolve(base, href, relation=None):
  if not is_absolute(base) or not is_well_formed(href):
    return None
  result = []
  type = PathType.RELATIVE_PATH
  (option, value) = href[0]
  if option == Option.HOST_IP:
    option = Option.HOST_NAME
  elif option == Option.PATH_TYPE:
    href = href[1:]
    type = value
    option = Option.PATH
  if option != Option.PATH or type == PathType.ABSOLUTE_PATH:
    _copy_until(base, result, option)
  else:
    _copy_until(base, result, Option.QUERY)
    if type == PathType.APPEND_RELATION:
      _append_and_normalize(result, Option.PATH,
                            format(relation, "x"))
      return result
    if type == PathType.RELATIVE_PATH:
      _remove_last_path_segment(result)
  _copy_until(href, result, Option.END)
  _append_and_normalize(href, Option.END, None)
  return result

def _copy_until(input, output, end):
  for (option, value) in input:
    if option >= end:
      break
    _append_and_normalize(output, option, value)

def _append_and_normalize(output, option, value):
  if option == Option.PATH:
    if value == ".":
      return
    if value == "..":
      _remove_last_path_segment(output)
      return
  elif option > Option.PATH:
    if len(output) >= 2 and                                     \
        output[-1] == (Option.PATH, "") and                     \
```

```
              (output[-2][0] < Option.PATH_TYPE or                     \
              output[-2] == (Option.PATH_TYPE, PathType.ABSOLUTE_PATH)):
            _remove_last_path_segment(output)
          if option >= Option.END:
            return
        output.append((option, value))

      def _remove_last_path_segment(output):
        if len(output) >= 1 and output[-1][0] == Option.PATH:
          del output[-1]
```

<CODE ENDS>

C.5. IRI Recomposition

This section defines how to recompose an IRI from a sequence of options that encodes an absolute IRI reference.

Applications MUST recompose an IRI from a sequence of options by using an algorithm that is functionally equivalent to the following Python 3.5 code.

To reduce variability, the hexadecimal notation when percent-encoding octets SHOULD use uppercase letters. The text representation of IPv6 addresses SHOULD follow the recommendations in Section 4 of RFC 5952 [RFC5952].

<CODE BEGINS>

```
      def recompose(href):
        if not is_absolute(href):
          return None
        result = ""
        no_path = True
        first_query = True
        for (option, value) in href:
          if option == Option.SCHEME:
            result += value + ":"
          elif option == Option.HOST_NAME:
            result += "//" + _encode_ireg_name(value)
          elif option == Option.HOST_IP:
            result += "//" + _encode_ip_address(value)
          elif option == Option.PORT:
            result += ":" + str(value)
          elif option == Option.PATH:
            result += "/" + _encode_path_segment(value)
            no_path = False
          elif option == Option.QUERY:
```

```
          if no_path:
            result += "/"
            no_path = False
          result += "?" if first_query else "&"
          result += _encode_query_argument(value)
          first_query = False
        elif option == Option.FRAGMENT:
          if no_path:
            result += "/"
            no_path = False
          result += "#" + _encode_fragment(value)
      if no_path:
        result += "/"
        no_path = False
      return result

    def _encode_ireg_name(s):
      return "".join(c if _is_ireg_name_char(c) else
          _encode_pct(c) for c in s)

    def _encode_ip_address(b):
      if len(b) == 4:
        return ".".join(str(c) for c in b)
      elif len(b) == 16:
        return "[" + ... + "]"  # see RFC 5952

    def _encode_path_segment(s):
      return "".join(c if _is_isegment_char(c) else
          _encode_pct(c) for c in s)

    def _encode_query_argument(s):
      return "".join(c if _is_iquery_char(c) and c != "&" else
          _encode_pct(c) for c in s)

    def _encode_fragment(s):
      return "".join(c if _is_ifragment_char(c) else
          _encode_pct(c) for c in s)

    def _encode_pct(s):
      return "".join(
          "%{0:0>2X}".format(c) for c in s.encode("utf-8"))

    def _is_ireg_name_char(c):
      return _is_iunreserved(c) or _is_sub_delim(c)

    def _is_isegment_char(c):
      return _is_ipchar(c)
```

```
      def _is_iquery_char(c):
        return _is_ipchar(c) or _is_iprivate(c) or c == "/" or c == "?"

      def _is_ifragment_char(c):
        return _is_ipchar(c) or c == "/" or c == "?"

      def _is_ipchar(c):
        return _is_iunreserved(c) or _is_sub_delim(c) or                \
               c == ":" or c == "@"

      def _is_iunreserved(c):
        return _is_alpha(c) or _is_digit(c) or                          \
               c == "-" or c == "." or c == "_" or c == "~" or          \
               _is_ucschar(c)

      def _is_alpha(c):
        return c >= "A" and c <= "Z" or c >= "a" and c <= "z"

      def _is_digit(c):
        return c >= "0" and c <= "9"

      def _is_sub_delim(c):
        return c == "!" or c == "$" or c == "&" or c == "'" or          \
               c == "(" or c == ")" or c == "*" or c == "+" or          \
               c == "," or c == ";" or c == "="

      def _is_ucschar(c):
        return c >= "\U000000A0" and c <= "\U0000D7FF" or              \
               c >= "\U0000F900" and c <= "\U0000FDCF" or              \
               c >= "\U0000FDF0" and c <= "\U0000FFEF" or              \
               c >= "\U00010000" and c <= "\U0001FFFD" or              \
               c >= "\U00020000" and c <= "\U0002FFFD" or              \
               c >= "\U00030000" and c <= "\U0003FFFD" or              \
               c >= "\U00040000" and c <= "\U0004FFFD" or              \
               c >= "\U00050000" and c <= "\U0005FFFD" or              \
               c >= "\U00060000" and c <= "\U0006FFFD" or              \
               c >= "\U00070000" and c <= "\U0007FFFD" or              \
               c >= "\U00080000" and c <= "\U0008FFFD" or              \
               c >= "\U00090000" and c <= "\U0009FFFD" or              \
               c >= "\U000A0000" and c <= "\U000AFFFD" or              \
               c >= "\U000B0000" and c <= "\U000BFFFD" or              \
               c >= "\U000C0000" and c <= "\U000CFFFD" or              \
               c >= "\U000D0000" and c <= "\U000DFFFD" or              \
               c >= "\U000E1000" and c <= "\U000EFFFD"

      def _is_iprivate(c):
        return c >= "\U0000E000" and c <= "\U0000F8FF" or              \
               c >= "\U000F0000" and c <= "\U000FFFFD" or              \
```

```
            c >= "\U00100000" and c <= "\U0010FFFD"
```

<CODE ENDS>

C.6. CoAP Encoding

This section defines how to construct CoAP options from an absolute, normalized, CBOR-encoded IRI Reference.

Applications MUST construct CoAP options by recomposing the sequence of options to an IRI (Appendix C.5 of this document), mapping the IRI to an URI (Section 3.1 of RFC 3987), and decomposing the URI into CoAP options (Section 6.4 of RFC 7252).

The following illustrative Python 3.5 code is roughly equivalent to this.

<CODE BEGINS>

```
def coap(href, to_proxy=False):
  if not is_absolute(href):
    return None
  result = b""
  previous = 0
  for (option, value) in href:
    if option == Option.SCHEME:
      pass
    elif option == Option.HOST_NAME:
      opt = 3  # Uri-Host
      val = value.encode("utf-8")
      result += _encode_coap_option(opt - previous, val)
      previous = opt
    elif option == Option.HOST_IP:
      opt = 3  # Uri-Host
      if len(value) == 4:
        val = ".".join(str(c) for c in b).encode("utf-8")
      elif len(value) == 16:
        val = b"[" + ... + b"]"  # see RFC 5952
      result += _encode_coap_option(opt - previous, val)
      previous = opt
    elif option == Option.PORT:
      opt = 7  # Uri-Port
      val = value.to_bytes((value.bit_length() + 7) // 8, "big")
      result += _encode_coap_option(opt - previous, val)
      previous = opt
    elif option == Option.PATH:
      opt = 11  # Uri-Path
      val = value.encode("utf-8")
```

```
          result += _encode_coap_option(opt - previous, val)
          previous = opt
        elif option == Option.QUERY:
          opt = 15  # Uri-Query
          val = value.encode("utf-8")
          result += _encode_coap_option(opt - previous, val)
          previous = opt
        elif option == Option.FRAGMENT:
          pass
      if to_proxy:
        (option, value) = href[0]
        opt = 39  # Proxy-Scheme
        val = value.encode("utf-8")
        result += _encode_coap_option(opt - previous, val)
        previous = opt
      return result

    def _encode_coap_option(delta, value):
      length = len(value)
      delta_nibble = _encode_coap_option_nibble(delta)
      length_nibble = _encode_coap_option_nibble(length)
      result = bytes([delta_nibble << 4 | length_nibble])
      if delta_nibble == 13:
        delta -= 13
        result += bytes([delta])
      elif delta_nibble == 14:
        delta -= 256 + 13
        result += bytes([delta >> 8, delta & 255])
      if length_nibble == 13:
        length -= 13
        result += bytes([length])
      elif length_nibble == 14:
        length -= 256 + 13
        result += bytes([length >> 8, length & 255])
      result += value
      return result

    def _encode_coap_option_nibble(n):
      if n < 13:
        return n
      elif n < 256 + 13:
        return 13
      elif n < 65536 + 256 + 13:
        return 14

  <CODE ENDS>
```

Author's Address

Klaus Hartke
Universitaet Bremen TZI
Postfach 330440
Bremen D-28359
Germany

Phone: +49-421-218-63905
Email: hartke@tzi.org