dnsop
Internet-Draft
Expires: April 18, 2015

B. Dickson

October 15, 2014

A Language to Describe the DNS Wire Format
draft-dickson-dnsop-spartacus-lang-00

Abstract

As part of the SPARTACUS DNS gateway system, building a full DNS parser was necessary. Parsing DNS packets is the only way to avoid propogating packets which are not correctly formatted DNS packets.

In order to facilitate building a new parser from scratch, the author chose to build a parser-builder which takes as input, a description of the DNS wire format.

This document describes the language created to facilitate this description, and includes the resulting DNS wire format description in this language.

Author's Note

Intended Status: Informational.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

DNS (The Domain Name System) has been around a long, long time. Over that time, the original Resource Record types have in some cases been made officially obsolete. Other, new Resource Records have been added. New definitions of bits in the header have arisen.

There have even been extensions added, which are intended to be backward compatible. The OPT pseudo-resource record, in particular,

overloads some of the standard field definitions in order to achieve its goals.

The end result is a wire format which is potentially difficult to parse.

In the interests of assisting future DNS endeavors, a complete description of the DNS wire format has been produced, and a comparitively simple language for facilitating this description has been created.

1.1. Rationale

Re-inventing the wheel, figuratively speaking, is frowned upon. By providing a description of the DNS wire format, and a language to accomplish this description, the author hopes that future work in the DNS arena might be made easier, at least in some cases.

The project which motivated this work, SPARTACUS (Secure, Private Aparatus for Resolution Transported Across Constraining and/or Unmaintained Systems), is intended to have multiple implementations in a variety of languages and environments. Creating a standard description of the DNS wire format, is intended to facilitate both an easier implementation effort, and a greater likelihood of compatible, interoprable implementations.

The SPARTACUS project is intended to create bidirectional DNS gateways for transporting DNS over other protocols and encodings, such as JSON over HTTP(S). This is intended to create "bridges" between DNS speakers. THe goal is to transport DNS messages from any DNS client implementation to any DNS server implementation. Each gateway needs to be liberal in what it accepts (any valid DNS message conforming to the relevant RFCs) and conservative in what it sends (only packets which parse correctly).

A secondary objective of the encoding in JSON is the use of the same names for data elements and structures as in the DNS RFCs. The idea is to provide human-readable JSON encodings, for easier diagnostics during development, and when investigating operational issues.

1.2. Related Work

A variety of other work exists, and provided inspiration for the SPARTACUS work. This includes web/JSON DNS portals, for providing DNS query responses in JSON format, often with a "looking glass" functionality.

- Multi-location DNS Looking Glass - Tool for performing DNS queries via RESTful interface in multiple locations, returning results in JSON format

- DNS Looking Glass - Tool for performing DNS queries via RESTful interface, returning results in JSON format

- DNS JSON - Source code project from circa 2009, partially developed but incomplete/abandoned

- DNSSEC-trigger[trigger] - embedded control function in NLnetlabs' Unbound resolver, for attempting DNS queries over TCP port 80 when DNSSEC problems are encountered

- Various other web-based DNS lookup tools

1.2.1. Comparison

There has been at least one previous effort to develop code for a DNS-JSON encoding, which appears to have been abandoned after one-way encoding was done, circa 2009. The project focused on presenting results to DNS queries in JSON format, with an intention to create a client gateway, which never materialized. The project can be found in two places ([JPF_jsondns] and [jsondns.org]). One major difference is that DNS query response status is converted to HTTP error codes, rather than being embedded in the JSON answer. This makes it unsuitable for bidirectional use. Only a few DNS type codes were implemented.

Another DNS JSON tool [fileformat.info], similarly focuses only on answers, with a limited number of type codes.

Yet another tool for looking up DNS via HTTP with JSON responses is the "dnsrest" [restdns.net]. It too focuses only on answer values, and is similarly not able to fully produce results that can be turned back into DNS answer packets.

The "DNS Looking Glass" [bortzmeyer.org], is primarily designed for returning DNS answer data. As such, it lacks encoding suitable for a bidirectional scheme. It is primarily focused on XML output, with JSON output organized around DNS resolution meta-data, plus answer data in a generic schema. (The schema itself is described in [draft-bortzmeyer-dns-json].)

The "Multilocation DNS Looking Glass" [dns-lg.com], uses a RESTful query mechanism of "node/qname/qtypename" to request the looking glass (LG) to perform a DNS lookup for the qname and qtype, and returns the response in a JSON format. The JSON format is generic, encapsulating all types as string data in presentation format, with a generic label of "rdata". This does not facilitate decoding easily, as the JSON scheme provides no information for parsing the rdata field. The type (qtype for the query, or type for answer/authority/additional) is in string (symbolic) form, and the elements are objects and thus in unordered lists. The JSON scheme is fine for

one-way encoding for human readability, but not suitable for two-way conversion back into DNS.

DNSSEC-trigger[trigger] can only be used in environments that use NLnetlabs' Unbound resolver, or where Unbound can be deployed as a replacement for existing recursive resolvers and/or stub resolvers.

A variety of other web lookup tools exist, predominantly producing DNS validation (zone structure and hierarchy), maps, meta-data, or literal output from the 'dig' tool, in formats as varied as the purposes of the tools. Dig output, while being reasonably deterministic, is not sufficiently well-formed as to facilitate "screen scraping" as a parsing method.

2. Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Syntax Overview

The syntax for the language is largely derived from only the abstract element types required to express data types and structures in DNS. In particular, the language has been kept as familiar and a simple as possible. Design choices were made to avoid over-abstracting elements which are by nature "difficult". Some objects have their size defined by other objects' values, or arithmetic expressions of values and literals. This includes array dimensions, and lengths of strings.

The general syntactic style uses braces ("{" and "}"), similar to the config files for BIND, for structural items.

Some familiarity with the DNS protocol is assumed.

3.1. Name Space

The name space of this language is tailored to the specific environment. Names need only be unique within their specific scope.

Since DNS messages are processed as "first in, first out" objects, the references to arrays have been simplified. Rather than keeping track of the index to an array, e.g. "a.b[3].c", the index is omitted, resulting in "a.b.c".

Relative object naming works the same as DNS search-list processing, depth-first. For example, while parsing "a.b.c.d", the name "foo"

would refer the first of the following that exists: "a.b.c.foo", "a.b.foo", "a.foo".

4. Syntax Elements

The syntactic elements of the language are base data types, structural elements, and preprocessing constructs. Additional elements provide the ability to annotate objects, and to define mnemonics for values.

4.1. Data Types

Base data types are encoded as "name:type", for a small number of predefined types and appropriate presentation formats:

- bitN - N-bit value, with 1-bit encoded as TRUE/FALSE.
- intN - unsigned N-bit integer, N is one of 8, 16, 32, 48.
- dotquad - IPv4 address.
- quadhex8 - IPv6 address.
- quadhex4 - 64-bit value (IPv6 prefix or IPv6 host-part).
- hex-string@EXPR - binary data of EXPR-specified length.
- base64@EXPR - binary data of EXPR-specified length.
- string - one or more character strings of 8-bit length and N-byte string.
- string? - optional single string.
- fqdn - Fully Qualified Domain Name.
- mbox - Mailbox: Fully Qualified Domain Name preceded by username. Wire format is identical to fqdn.

Example of a base64 object named "MAC" whose size is specified by "MACsize", in context:


```
TSIG (RFC 2845) {
  Algorithm:fqdn
  TimeSigned:int48
  Fudge:int16
  MACsize:int16
  MAC:base64@MACsize
  OriginalID:int16
  Error:int16
```

```
  OtherSize:int16
  OtherData:base64@OtherSize
  }
```

When the parser decodes MACsize (e.g. as the unsigned integer "12"), it would then decode 12 bytes of data into MAC, and convert that data into base-64 (for encoding to JSON).

4.2. Enumeration and RFC References

The remainder of the elements of the language exist to permit annotation of well-known values (such as "NXDOMAIN" for RCODE=3), and for providing human-friendly RFC references. These are:

(RFC XXXX) - for any object name, associates a given RFC with it. Added by the parser to the corresponding JSON string.

Enum ENUM_NAME - allows for integer types, to have mnemonics associated with them, as "value:name" pairs.

Of ENUM_NAME - adds the name whenver the corresponding value is parsed (for this integer-type object).

Example of RFC:

```
NSID (RFC 5001) {
  NSIDContent:hex-string@*Len
  }
```

When processing an NSID, the JSON string would be:

"NSID (RFC 5001)"

instead of

"NSID"

Example of enum:


```
enum classes {
```

```
 1:IN
 3:CH
 254:NONE
 255:ANY
 }
```

CLASS:int16 of classes

When processing a CLASS object with value 1, the JSON encoding would be:

```
   "CLASS" : [ "IN" : 1 ]
```

instead of

```
   "CLASS" : 1
```

4.3. Structural Elements

There are two structural elements:

- Structure - an ordered group of elements, including any combination of Data Types and further Structural elements. There is no limit on structure depth.

- Array - an Array has one child type, which can be either a Data Type, or simple Structure. The size of the array can be explicit, referencing the name of any integer type, or implicit, referencing the name of an integer whose value is the total length of the array.

- Switch and Case - similar to the C language elements, these provide a way of encoding a sparse array. The Switch object has a child Structure which consists only of Case objects. Each Case object associates a value with a named object (Structure or Data Type). The Switch references the name of an integer object type, which is compared to the Case objects (when parsing data).

Example of simple structure:

```
  HFlags {
   QR:bit1
   Opcode:bit4
   AA:bit1
   TC:bit1
   RD:bit1
   RA:bit1
   Z:bit1
   AD:bit1
   CD:bit1
   RCODE:bit4
   }
```

Example of an array DAU_TYPES, in context:

```
  LIST_LENGTH:int16
  DAU_TYPES: array[LIST_LENGTH] DAU_TYPE {
    ALG_CODE:int8
    }
```

Example of switch Field3 based on object TYPE, and corresponding cases ("case *" is equivalent to "default" in C):

```
  TYPE:int16 of rrtype
  Field3: switch TYPE {
    case 41: UDPSIZEFIELD {
     UDPSIZE:int16
     }
    case *: CLASSFIELD {
     CLASS:int16
     }
    }
```

When parsing data, if the TYPE had value 41, when converted to JSON, the object name "UDPSIZE" would appear, rather than "CLASS".

4.4. Preprocessing Elements

There are two elements which provide preprocessing capabilities:

- Define - objects are ways of doing logical cut/paste of input file contents.

- Reference - to the same named object created with a "define", results in a literal copy of the original range of file contents.

This operates much like "#define" does in the C language. By doing this, identical structures and object types which occur in different

places can be maintained in one section of the file. In partucular, the Resource Records from all three sections can be defined once.

Example of one define and two references to RDATATYPE. Note that an object named RDLENGTH must be present in an ancestor of both parent objects:

Input file before preprocessor:

```
define RDATATYPE {
 case 1: A {
  Address:dotquad
  }
... (lots of lines omitted for clarity)
 case 256: URI {
  GENERIC_RDATA:hex-string@RDLENGTH
  }
 }

 Answer {
  RR_LIST: array[HEADER.ANCOUNT] RR {
   NAME:fqdn
   TYPE:int16 of rrtype
   CLASS:int16 of classes
   TTL:int32
   RDLENGTH:int16
   RDATA: switch TYPE {
    reference RDATATYPE
    }
   }
  }
 Authority {
  RR_LIST: array[HEADER.NSCOUNT] RR {
   NAME:fqdn
   TYPE:int16 of rrtype
   CLASS:int16 of classes
   TTL:int32
   RDLENGTH:int16
   RDATA: switch TYPE {
```

```
          reference RDATATYPE
          }
         }
        }

     Result of preprocessing:

      Answer {
       RR_LIST: array[HEADER.ANCOUNT] RR {
```



```
        NAME:fqdn
        TYPE:int16 of rrtype
        CLASS:int16 of classes
        TTL:int32
        RDLENGTH:int16
        RDATA: switch TYPE {
          case 1: A {
           Address:dotquad
           }
         ... (lots of lines omitted for clarity)
          case 256: URI {
           GENERIC_RDATA:hex-string@RDLENGTH
           }
         }
        }
       }
      Authority {
       RR_LIST: array[HEADER.NSCOUNT] RR {
        NAME:fqdn
        TYPE:int16 of rrtype
        CLASS:int16 of classes
        TTL:int32
        RDLENGTH:int16
        RDATA: switch TYPE {
          case 1: A {
           Address:dotquad
           }
         ... (lots of lines omitted for clarity)
          case 256: URI {
           GENERIC_RDATA:hex-string@RDLENGTH
           }
         }
```

```
        }
      }
```

5. Interactions and Behavior

6. Security Considerations

None per se.

7. IANA Considerations

This document contains no IANA-specific material.

8. Acknowledgements

To be added later.

9. References

9.1. Normative References

[RFC1033] Lottor, M., "Domain administrators operations guide", RFC 1033, November 1987.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, November 1987.

[RFC2136] Vixie, P., Thomson, S., Rekhter, Y., and J. Bound, "Dynamic Updates in the Domain Name System (DNS UPDATE)", RFC 2136, April 1997.

[RFC2181] Elz, R. and R. Bush, "Clarifications to the DNS Specification", RFC 2181, July 1997.

[RFC2308] Andrews, M., "Negative Caching of DNS Queries (DNS

NCACHE)", RFC 2308, March 1998.

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, March 2005.

[RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, March 2005.

[RFC4035] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Protocol Modifications for the DNS Security Extensions", RFC 4035, March 2005.

[RFC5011] StJohns, M., "Automated Updates of DNS Security (DNSSEC) Trust Anchors", STD 74, RFC 5011, September 2007.

[RFC5155] Laurie, B., Sisson, G., Arends, R., and D. Blacka, "DNS Security (DNSSEC) Hashed Authenticated Denial of Existence", RFC 5155, March 2008.

9.2. Informative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[JPF_jsondns]
"DNS over HTTP", <http://github.com/jpf/jsondns>.

[jsondns.org]
Franusic, J., "Query DNS via REST", <http://jsondns.org/>.

[fileformat.info]
Marcuse, A., "DNS in client-side JavaScript", <http://www.fileformat.info/tool/rest/dns-json.htm>.

[restdns.net]
"REST-DNS", <http://restdns.net/>.

[bortzmeyer.org]
Bortzmeyer, S., "DNS Looking Glass",
<http://www.bortzmeyer.org/dns-lg.html>.

[draft-bortzmeyer-dns-json]
Bortzmeyer, S., "DNS in JSON",
<http://tools.ietf.org/html/draft-bortzmeyer-dns-json-01>.

[dns-lg.com]
Cambus, F., "Multilocation DNS Looking Glass",
<http://www.dns-lg.com/>.

[trigger] NLnet Labs, "Dnssec-Trigger",
<http://www.nlnetlabs.nl/projects/dnssec-trigger/>.

Appendix A. DNS Message Format Encoding

The entire encoding of the DNS message format follows.

```
# opcodes
enum opcodes {
 0:Query
 2:Status
 4:Notify
 5:Update
 }

# classes
enum classes {
 1:IN
```



```
 3:CH
 254:NONE
 255:ANY
 }

enum ednstype {
 3:NSID
 5:DAU
 6:DHU
 7:N3U
 }
```

```
enum rrtype {
 1:A
 28:AAAA
 15:MX
 2:NS
 12:PTR
 5:CNAME
 39:DNAME
 6:SOA
 37:CERT
 48:DNSKEY
 43:DS
 32769:DLV
 47:NSEC
 50:NSEC3
 51:NSEC3PARAM
 46:RRSIG
 24:SIG
 52:TLSA
 44:SSHFP
 249:TKEY
 250:TSIG
 35:NAPTR
 33:SRV
 99:SPF
 16:TXT
 18:AFSDB
 19:X25
 17:RP
 21:RT
 20:ISDN
 29:LOC
 27:GPOS
 104:NID
 105:L32
 106:L64
 107:LP
```



```
 251:IXFR
 252:AXFR
 253:MAILB
 254:MAILA
```

```
 255:*
 256:URI
 257:CAA
 32768:TA
 41:OPT
 }

#
# EDNS Option-codes follow
define EDNSTYPE {
 case 3: NSID (RFC 5001) {
  NSIDContent:hex-string@*Len
  }
 case 5: DAU (RFC 6975) {
  LIST_LENGTH:int16
  DAU_TYPES: array[LIST_LENGTH] DAU_TYPE {
    ALG_CODE:int8
    }
  }
 case 6: DHU (RFC 6975) {
  LIST_LENGTH:int16
  DHU_TYPES: array[LIST_LENGTH] DHU_TYPE {
    ALG_CODE:int8
    }
  }
 case 7: N3U (RFC 6975) {
  LIST_LENGTH:int16
  N3U_TYPES: array[LIST_LENGTH] NSU_TYPE {
    ALG_CODE:int8
    }
  }
 }

define RDATATYPE {
 case 1: A {
  Address:dotquad
  }
 case 28: AAAA {
  Address:quadhex8
  }
 case 15: MX {
  Preference:int16
  MailExchanger:fqdn
  }
```

```
    case 2: NS {
     Target:fqdn
     }
    case 12: PTR {
     Target:fqdn
     }
    case 5: CNAME {
     Target:fqdn
     }
    case 39: DNAME {
     Target:fqdn
     }
    case 6: SOA {
     MasterServerName:fqdn
     MaintainerName:mbox
     Serial:int32
     Refresh:int32
     Retry:int32
     Expire:int32
     NegativeTtl:int32
     }
    case 37: CERT (RFC 4398) {
     Type:int16
     KeyTag:int16
     Algorithm:int8
     Data:base64@RDLENGTH-5
     }
    case 48: DNSKEY (RFC 4034) {
     Flags:int16
     protocol:int8=3
     Algorithm:int8
     data:base64@RDLENGTH-4
     #Tag:int16=%#(derived/calculated/optional?)
     }
    case 43: DS (RFC 4034) {
     Keytag:int16
     Algorithm:int8
     DigestType:int8
     DelegationKey:hex-string@RDLENGTH-4
     }
    case 32769: DLV {
     Keytag:int16
     Algorithm:int8
     DigestType:int8
     DelegationKey:hex-string@RDLENGTH-4
     }
    case 47: NSEC (RFC 4034) {
```

```
     NextName:fqdn
```



```
     FlagBits:hex-string@*RDLENGTH
     }
    case 50: NSEC3 (RFC 5155) {
     Algorithm:int8
     Flags {
      ResvBits:bit7
      OptOut:bit1
      }
     Iterations:int16
     SaltLength:int8
     Salt:hex-string@SaltLength
     HashLength:int8
     NextHash:hex-string@HashLength
     FlagBits:hex-string@RDLENGTH-6-SaltLength-HashLength
     }
    case 51: NSEC3PARAM (RFC 5155) {
     Algorithm:int8
     Flags:int8
     Iterations:int16
     SaltLength:int8
     Salt:hex-string@SaltLength
     }
    case 46: RRSIG (RFC 4034) {
     Type:int16
     Algorithm:int8
     Labels:int8
     OTTL:int32
     SigExp:int32
     SigInc:int32
     Tag:int16
     Signer:fqdn
     Sig:base64@*RDLENGTH
     }
    case 24: SIG (RFC 2931) {
     Type:int16
     Algorithm:int8
     Labels:int8
     OTTL:int32
     SigExp:int32
     SigInc:int32
```

```
     Tag:int16
     Signer:fqdn
     Sig:base64@*RDLENGTH
     }
    case 52: TLSA (RFC 6698) {
     CertUsage:int8
     Selector:int8
     MatchType:int8
```



```
     Data:hex-string@RDLENGTH-3
     }
    case 44: SSHFP (RFC 4255) {
     Algorithm:int8
     DigestType:int8
     Fingerprint:hex-string@RDLENGTH-2
     }
    case 249: TKEY (RFC 2930) {
     Algorithm:fqdn
     Incep:int32
     Exp:int32
     Mode:int16
     Error:int16
     KeySize:int16
     KeyData:hex-string@KeySize
     OtherSize:int16
     OtherData:hex-string@OtherSize
     }
    case 250: TSIG (RFC 2845) {
     Algorithm:fqdn
     TimeSigned:int48
     Fudge:int16
     MACsize:int16
     MAC:base64@MACsize
     OriginalID:int16
     Error:int16
     OtherSize:int16
     OtherData:base64@OtherSize
     }
    case 35: NAPTR (RFC 3403) {
     Order:int16
     Preference:int16
     Flags:string
```

```
     Services:string
     Regexp:string
     Replacement:fqdn
     }
    case 33: SRV (RFC 2782) {
     Port:int16
     Priority:int16
     Weight:int16
     Server:fqdn
     }
    case 99: SPF (RFC 4408) {
     Text:string@*RDLENGTH
     }
    case 16: TXT {
     Text:string@*RDLENGTH
```



```
     }
    case 41: OPT (RFC 6891) {
     TLV_LIST: array[*RDLENGTH] TLV {
      TYPE:int16 of ednstype
      Len:int16
      Data: switch TYPE {
       reference EDNSTYPE
       }
      }
     }
    ###
    ### Obsolete Stuff Begins
    ###
    ## AFS & X25 stuff Begins
    case 18: AFSDB (RFC 1183) {
     SubType:int16
     Hostname:fqdn
     }
    case 19: X25 (RFC 1183) {
     PSDN:string
     }
    case 17: RP (RFC 1183) {
     Who:mbox
     What:fqdn
     }
    case 21: RT (RFC 1183) {
```

```
     Preference:int16
     Via:fqdn
     }
    case 20: ISDN (RFC 1183) {
     Number:string
     SA:string?@*RDLENGTH
     }
    ## X25 Stuff Ends
    ## Other Obsolete Stuff
    case 29: LOC (RFC 1876) {
     Version:int8
     Size:int8
     HorPrec:int8
     VertPrec:int8
     Longitude:int32
     Latitude:int32
     Altitude:int32
     }
    case 27: GPOS (RFC 1712) {
     Long:string
     Lat:string
     Alt:string
```



```
     }
    ###
    ### ILNP Stuff
    ###
    case 104: NID (RFC 6742) {
     Pref:int16
     Node:quadhex4
     }
    case 105: L32 (RFC 6742) {
     Pref:int16
     ID:dotquad
     }
    case 106: L64 (RFC 6742) {
     Pref:int16
     ID:quadhex4
     }
    case 107: LP (RFC 6742) {
     Pref:int16
     Target:fqdn
```

```
     }
    ##
    ## Basically unsupported types follow
    case 251: IXFR {
     GENERIC_RDATA:hex-string@RDLENGTH
     }
    case 252: AXFR {
     GENERIC_RDATA:hex-string@RDLENGTH
     }
    case 253: MAILB {
     GENERIC_RDATA:hex-string@RDLENGTH
     }
    case 254: MAILA {
     GENERIC_RDATA:hex-string@RDLENGTH
     }
    case 255: * {
     GENERIC_RDATA:hex-string@RDLENGTH
     }
    case 256: URI {
     GENERIC_RDATA:hex-string@RDLENGTH
     }
    case 257: CAA {
     GENERIC_RDATA:hex-string@RDLENGTH
     }
    case 32768: TA {
     GENERIC_RDATA:hex-string@RDLENGTH
     }

    }
```



```
   # Draft JSON typenames and element names/types
   PACKET (RFC 1035) {
    Header {
     ID:int16
     HFlags {
      QR:bit1
      Opcode:bit4 of opcodes
      AA:bit1
      TC:bit1
      RD:bit1
      RA:bit1
      Z:bit1=0
```

```
      AD:bit1
      CD:bit1
      RCODE:bit4
      }
     QDCOUNT:int16
     ANCOUNT:int16
     NSCOUNT:int16
     ARCOUNT:int16
     }
    Question {
     QContinuum: switch PACKET.Header.HFlags.Opcode {
       case 0: QUESTION (RFC 1035) {
        QNAME:fqdn
        QTYPE:int16
        QCLASS:int16 of classes
        }
       case 4: NOTIFY (RFC 1996) {
        QNAME:fqdn
        QTYPE:int16=SOA
        QCLASS:int16 of classes
        }
   # NB:
   # Opcode=UPDATE: Redefines Names & Semantics of sections as follows:
   #  ZONE
   #  Prerequisite
   #  Update
   #  Additional_Data
   #  (All sections may have data, even though QR=0)
   #
       case 5: ZONE (RFC 2136) {
        ZNAME:fqdn
        ZTYPE:int16=SOA
        ZCLASS:int16 of classes
        }
       }
     }
```



```
    Answer {
     RR_LIST: array[HEADER.ANCOUNT] RR {
      NAME:fqdn
      TYPE:int16 of rrtype
      CLASS:int16 of classes
```

```
    TTL:int32
    RDLENGTH:int16
    RDATA: switch TYPE {
     reference RDATATYPE
     }
    }
   }
  Authority {
   RR_LIST: array[HEADER.NSCOUNT] RR {
    NAME:fqdn
    TYPE:int16 of rrtype
    CLASS:int16 of classes
    TTL:int32
    RDLENGTH:int16
    RDATA: switch TYPE {
     reference RDATATYPE
     }
    }
   }
  Additional {
   RR_LIST: array[HEADER.ARCOUNT] RR {
    NAME:fqdn
    TYPE:int16 of rrtype
    # do overload on CLASS and TTL for TYPE=41 (OPT)
    Field3: switch TYPE {
     case 41: UDPSIZEFIELD {
      UDPSIZE:int16
      }
     case *: CLASSFIELD {
      CLASS:int16 of classes
      }
     }
    Field4: switch TYPE {
     case 41: Extended_RCode_Flags {
      RCode:bit8
      Version:bit8
      DO:bit1
      Resv:bit15
      }
     case *: TTLFIELD {
      TTL:int32
      }
     }
```

```
      RDLENGTH:int16
      RDATA: switch TYPE {
       reference RDATATYPE
       }
      }
     }
    }
```

Author's Address

Brian Dickson
12047B 36th Ave NE
Seattle, WA 98125

Email: brian.peter.dickson@gmail.com