NAME

udns - stub DNS resolver library

SYNOPSYS

#include <udns.h>
struct **dns_ctx**;
struct **dns_query**;
extern struct dns_ctx **dns_defctx**;
struct dns_ctx **ctx*;
typedef void **dns_query_fn**(*ctx*, void **result*, void **data*);
typedef int
dns_parse_fn(const unsigned char **qnd*,
const unsigned char **pkt*,
const unsigned char **cur*,
const unsigned char **end*,
void ***resultp*);

cc ... -l**udns**

DESCRIPTION

The DNS library, **udns**, implements thread-safe stub DNS resolver functionality, which may be used both traditional, syncronous way and asyncronously, with application-supplied event loop.

While DNS works with both TCP and UDP, performing UDP query first and if the result does not fit in UDP buffer (512 bytes max for original DNS protocol), retrying the query over TCP, the library uses UDP only, but uses EDNS0 (RFC2671) extensions which allows larger UDP buffers.

The library uses single UDP socket to perform all operations even when asking multiple nameservers. This way, it is very simple to use the library in asyncronous event-loop applications: an application should add only single socket to the set of filedescriptors it monitors for I/O.

The library uses two main objects, *resolver context* of type **struct dns_ctx**, and *query structure* of type **struct dns_query**, both are opaque for an application. Resolver context holds global information about the resolver, such as list of nameservers to use, list of active requests and the like. Query objects holds information about a single DNS query in progress and are allocated/processed/freed by the library.

Pointer to query structure may be treated as an identifier of an in-progress query and may be used to cancel the asyncronous query or to wait for it to complete.

Asyncronous interface works as follows. An application initializes resolver context, submits any number of queries for it using one of supplied **dns_submit**_*XXX*() routines (each return the query identifier as pointer to query structure), waits for input on the UDP socket used by the library, and gives some control to the library by calling **dns_ioevent**() and **dns_timeouts**() routines when appropriate. The library performs all necessary processing and executes application supplied callback routine when a query completes (either successefully or not), giving it the result if any, pointer to the resolver context (from which completion status may be obtained), and the data pointer supplied by an application when the query has been submitted. When submitting a query, an application requests how to handle the reply -- to either return raw DNS reply packet for its own low-level processing, or it may provide an address of *parsing routine* of type **dns_parse_fn** to perform conversion of on-wire format into easy to use data structure (the library provides parsing routines for several commonly used resource record types, as well as type-safe higher-level inteface that requests parsing automatically). The I/O monitoring and timeout handling may be either traditional select() or poll() based, or any callback-driven technique may be used.

Additionally, the library provides traditional syncronous interface, which may be intermixed with asyncronous calls (during syncronous query processing, other asyncronous queries for the same resolver context continued to be processed as usual). An application uses one of numerous **dns_resolve**_*XXX*() routines provided by the library to perform a query. As with asyncronous interface, an application may either request to return raw DNS packet or type-specific data structure by providing the parsing routine to handle the reply. Every routine from **dns_resolve**_*XXX*() series return pointer to result or NULL in case of any error. Query completion status (or length of the raw DNS packet) is available from the resolver context using **dns_status**() routine, the same way as for the asyncronous interface.

Internally, library uses on-wire format of domain names, referred to as *DN format* in this manual page. This is a series of domain *labels* whith preceeding length byte, terminated by zero-length label wich is integral part of the DN format. There are several routines provided to convert from traditional asciiz string to DN and back. Higher-level type-specific query interface hides the DN format from an application.

COMMON DEFINITIONS

Every DNS Resource Record (RR) has a *type* and a *class*. The library defines several integer constants,

DNS_C_*XXX* and **DNS_T**_*XXX*, to use as symbolic names for RR classes and types, such as **DNS_C_IN** for Internet class, **DNS_T_A** for IPv4 address record type and so on. See udns.h header file for complete list of all such constants.

The following constants are defined in udns.h header file:

DNS_MAXDN (255 bytes)
Maximum length of the domain name in internal (on-wire) DN format.

DNS_MAXLABEL (63 bytes)
Maximum length of a single label in DN format.

DNS_MAXNAME (1024 bytes)
Maximum length of asciiz format of a domain name.

DNS_HSIZE (12 bytes)
Size of header in DNS packet.

DNS_PORT (53)
Default port to use when contacting a DNS server.

DNS_MAXSERV (6 servers)
Maximum number of DNS servers to use.

DNS_MAXPACKET (512 bytes)
Maximum length of DNS UDP packet as specified by original DNS protocol

DNS_EDNS0PACKET (4096 bytes)
Default length of DNS UDP packet (with EDNS0 extensions) the library uses. Note that recursive nameservers usually resides near the client asking them to resolve names, e.g. on the same LAN segment or even on the same host, so UDP packet fragmentation isn't a problem in most cases. Note also that the size of actual packets will be as many bytes as actual reply size requires, which is smaller than this value in almost all cases.

Additionally, several constants are defined to simplify work with raw DNS packets, such as DNS response codes (**DNS_R**_*XXX*), DNS header layout (**DNS_H**_*XXX*) and others. Again, see udns.h for complete list. Library error codes (**DNS_E**_*XXX*) are described later in this manual page.

RESOLVER CONTEXT

Resolver context, of type **struct dns_ctx**, is an object which is opaque to an application. Several routines provided by the library to initialize, copy and free resolver contexts. Most other high-level routines in this library expects a pointer to resolver context, *ctx*, as the first argument. There is a default resolver context available, named **dns_defctx**. When the context pointer *ctx* passed to a routine is NULL, **dns_defctx** is used. Several resolver contexts may be active at the same time, for example, when an application is multi-threaded and each thread uses resolver.

In order to use the library, an application should initialize and open one or more resolver context objects. These are two separate actions, performed by **dns_init**() (or **dns_reset**()), and **dns_open**(). Between the two calls, an application is free to pefrom additional initialisation, such as setting custom nameservers, options or domain search lists. Optionally, in case no additional custom initialisation is required, **dns_init**() may open the context if *do_open* argument (see below) is non-zero.

When initializing resolver context, the library uses information from system file /etc/resolv.conf (see **resolv.conf**(5)), consults environment variables **$LOCALDOMAIN**, **$NSCACHEIP**, **$NAMESERVERS** and **$RES_OPTIONS**, and local host name to obtain list of local nameservers, domain name search list and various resolver options.

The following routines to initialize resolver context are available:

void **dns_reset**(*ctx*)
int **dns_init**(*ctx*, int *do_open*)

dns_reset() resets a given resolver context to default values, preparing it to be opened by **dns_open**(). It is ok to call this routine against opened and active context - all active queries will be dropped, sockets will be closed and so on. This routine does not initialize any parameters from system configuration files, use **dns_init**() for this. There's no error return - operation always succeeds. **dns_init**() does everything **dns_reset**() does, plus initializes various parameters of the context according to system configuration and process environment variables. If *do_open* is non-zero, **dns_init**() calls *dns_open*(), so that the whole library initialisation is performed in a single step.

struct dns_ctx ***dns_new**(struct dns_ctx **copy*)
void **dns_free**(*ctx*)

dns_new() allocates new resolver context and copies all parameters for a given resolver context *copy*, or default context if *copy* is NULL, and returns pointer to the newly allocated context. The context being copied should be initialized. **dns_new**() may fail if there's no memory available to make a copy of *copy*, in which case the routine will return NULL pointer. **dns_free**() is used to close assotiated socket and free resolver context resources and cancelling (abandoming) all active queries assotiated with it. It's an error to free **dns_defctx**, only dynamically allocated contexts

returned by **dns_new**() are allowed to be freed by **dns_free**().

int **dns_add_serv**(*ctx*, const char **servaddr*)
int **dns_add_serv_s**(*ctx*, const struct sockaddr **sa*)
int **dns_add_srch**(*ctx*, const char **srch*)
Add an element to list of nameservers (**dns_add_serv**(), as asciiz-string *servaddr* with an IP address of the nameserver, and **dns_add_serv_s**(), as initialized socket address *sa*), or search list (**dns_add_srch**(), as a pointer to domain name) for the given context *ctx*. If the last argument is a NULL pointer, the corresponding list (search or nameserver) is reset instead. Upon successeful completion, each routine returns new number of elements in the list in question. On error, negative value is returned and global variable **errno** is set appropriately. It is an error to call any of this functions if the context is opened (after **dns_open**() or **dns_init**() with non-zero argument).

int **dns_set_opts**(*ctx*, const char **opts*)
set resolver context options from *opts* string, in the same way as processing **options** statement in resolv.conf and **$RES_OPTIONS** environment variable. Return number of unrecognized/invalid options found (all recognized and valid options gets processed).

void **dns_set_opt**(*ctx*, int *opt*, *val*)
TODO The *flags* argument is a bitmask with the following bits defined:

DNS_NOSRCH
do not perform domain name search in search list.

DNS_NORD
do not request recursion when performing queries (i.e. don’t set RD flag in querues).

DNS_AAONLY
request authoritative answers only (i.e. set AA flag in queries).

int **dns_open**(*ctx*)
int **dns_sock**(const *ctx*)
void **dns_close**(*ctx*)
dns_open() opens the UDP socket used for queries if not already open, and return assotiated filedescriptor (or negative value in case of error). Before any query can be submitted, the context should be opened using this routine. And before opening, the context should be initialized. **dns_sock**() return the UDP socket if open, or -1 if not. **dns_close**() closes the UDP socket if it was open, and drops all active queries if any.

int **dns_active**(const *ctx*)

return number of active queries queued for the given context *ctx*, or zero if none.

int **dns_status**(const *ctx*)

return status code from last operation. When using syncronous interface, this is the query completion status of the last query. With asyncronous interface, from within the callback routine, this is the query completion status of the query for which the callback is being called. When query submission fails, this is the error code indicating failure reason. All error codes are negative and are represented by **DNS_E**_*XXX* constants described below.

void **dns_ioevent**(*ctx*, time_t *now*)

this routine may be called by an application to process I/O events on the UDP socket used by the library, as returned by **dns_sock**(). The routine tries to receive incoming UDP datagram from the socket and process it. The socket is set up to be non-blocking, so it is safe to call the routine even if there's no data to read. The routine will process as many datagrams as are queued for the socket, so it is safe to use it with either level-triggered or edge-triggered I/O monitoring model. The *now* argument is either a current time as returned by **time**(), or 0, in which case the routine will obtain current time by it's own.

int **dns_timeouts**(*ctx*, int *maxwait*, time_t *now*)

process any pending timeouts and return number of secounds from current time (*now* if it is not 0) to the time when the library wants the application to pass it control to process more queued requests. In case when there are no requests pending, this time is -1. The routine will not request a time larger than *maxwait* secounds if it is greather or equal to zero. If *now* is 0, the routine will obtain current time by it's own; when it is not 0, it should contain current time as returned by **time**().

typedef void **dns_utm_fn**(*ctx*, int *timeout*, void **data*)

void **dns_set_tmcbck**(*ctx*, dns_utm_fn **utmfn*, void **data*)

An application may use custom callback-based I/O multiplexing mechanism. Usually such a mechanism have concept of a *timer*, and an ability to register a timer event in a form of a callback routine which will be executed after certain amount of time. In order to use such an event mechanism, udns provides an ability to register and de-register timer events necessary for internal processing using whatever event mechanism an application uses. For this to work, it is possible to assotiate a pointer to a routine that will perform necessary work for (de)registering timer events with a given resolver context, and udns will call that routine at appropriate times. Prototype of

such a routine is shown by **dns_utm_fn** typedef above. Libudns assotiates single timer with resolver context. User-supplied *utmfn* routine will be called by the library with the following arguments:

ctx == NULL
: delete user timer, at context free time or when an application changes user timer request routine using **dns_set_tmcbck**();

ctx != NULL, *timeout* < 0
: don't fire timer anymore, when there are no active requests;

ctx != NULL, *timeout* == 0
: fire timer at the next possibility, but not immediately;

ctx != NULL, *timeout* > 0
: fire timer after *timeout* seconds after now.

The *data* argument passed to the routine will be the same as passed to **dns_set_tmcbck**().

When a timer expires, an application should call **dns_timeouts**() routine (see below). Non-callback timer usage is provided too.

XXXX TODO: some more resolver context routines, like dns_set_dbgfn() etc.

QUERY INTERFACE

There are two ways to perform DNS queries: traditional syncronous way, when udns performs all the necessary processing and return control to the application only when the query completes, and asyncronous way, when an application submits one or more queries to the library using given resolver context, and waits for completion by monitoring filedescriptor used by library and calling library routines to process input on that filedescriptor. Asyncronous mode works with callback routines: an application supplies an address of a routine to execute when the query completes, and a data pointer, which is passed to the callback routine.

Queries are submitted to the library in a form of **struct dns_query**. To perform asyncronous query, an application calls one of the **dns_submit**_*XXX*() rounines, and provides necessary information for a callback, together with all the query parameters. When the query completes, library will call application-supplied callback routine, giving it the resolver context (wich holds query completion

status), dynamically allocated result (which will be either raw DNS packet or, if applicatin requested parsing the result by specifying non-NULL parse routine, ready-to-use type-specific structure), and a data pointer provided by an application when it submitted the query. It is the application who's responsible for freeing the result memory.

Generic query callback routine looks like this:
typedef void
dns_query_fn(*ctx*, void ***result*, void ***data*)
Type-specific query interface expects similar form of callback routine with the only difference in type of **result** argument, which will be pointer to specific data structure (decoded reply) instead of this void pointer to raw DNS packet data.

Result parsing routine looks like this:
typedef int
dns_parse_fn(const unsigned char ***qdn*,
const unsigned char ***pkt*,
const unsigned char ***cur*,
const unsigned char ***end*,
void ****resultp*);
When called by the library, the arguments are as follows: *pkt* points to the start of the packet received; *end* points past the end of the packet received; *cur* points past the query DN in the query section of the packet; *qdn* points to the original query DN. The routine should allocate a single buffer to hold the result, parse the reply filling in the buffer, and return the buffer using *resultp* argument. It returns 0 in case of error, or udns error code (**DNS_E**_*XXX* constants) in case of error. Note that by the time when the parse routine is called by the library, packet is already verified to be a reply to the original query, by matching query DN, query class and query type.

Type-specific query inteface supplies necessary parsing routines automatically.

In case of error, query completion status as returned by **dns_status**(*ctx*), will contain one of the following values:

positive value
length of raw DNS packet if parsing is not requested.

0 the query was successeful and the *reply* points to type-specific data structure.

DNS_E_TEMPFAIL
temporary error, the resolver nameserver was not able to process our query or timed out.

DNS_E_PROTOCOL
protocol error, a nameserver returned malformed reply.

DNS_E_NXDOMAIN
the domain name does not exist.

DNS_E_NODATA
there is no data of requested type found.

DNS_E_NOMEM
out of memory while processing request.

DNS_E_BADQUERY
some aspect of the query (most common is the domain name in question) is invalid, and the library can’t even start a query.

Library provides two series of routines which uses similar interface -- one for asyncronous queries and another for syncronous queries. There are two general low-level routines in each series to submit (asyncronous interface) and resolve (syncronous interface) queries, as well as several type-specific routines with more easy-to-use interfaces. To submit an asyncronous query, use one of **dns_submit**_*XXX*() routine, each of which accepts query parameters, pointers to callback routine and to callback data, and optional current time hint. Note type-specific **dns_submit**_*XXX*() routines expects specific type of the callback routine as well, which accepts reply as a pointer to corresponding structure, not a void pointer). Every **dns_submit**_*XXX*() routine return pointer to internal query structure of type struct dns_query, used as an identifier for the given query.

To resolve a query syncronously, use one of **dns_resolve**_*XXX*() routines, which accepts the same query parameters (but not the callback pointers) as corresponding **dns_submit**_*XXX*(), and return the query result, which is the same as passed to the callback routine in case of asyncronous interface.

In either case, the result memory (if the query completed successefully) is dynamically allocated and should be freed by an application. If the query failed for any reason, the result will be NULL, and error status will be available from **dns_status**(*ctx*) routine as shown above.

struct dns_query *
dns_submit_dn(*ctx*,
const unsigned char ***dn*, *qcls*, *qtyp*, *flags*,
parse, *cbck*, *data*)
struct dns_query *
dns_submit_p(*ctx*,
const char ***name*, *qcls*, *qtyp*, *flags*,
parse, *cbck*, *data*)
enum dns_class *qcls*;
enum dns_type *qtyp*;
int *flags*;
dns_parse_fn ***parse*;
dns_query_fn ***cbck*;
void ***data*;

submit a query for processing for the given resolver context *ctx*. Two routines differs only in 3rd argument, which is domain name in DN format (*dn*) or asciiz string (*name*). The query will be performed for the given domain name, with type *qtyp* in class *qcls*, using option bits in *flags*, using RR parsing routine pointed by *parse* if not-NULL, and upon completion, *cbck* function will be called with the *data* argument. In case of successeful query submission, the routine return pointer to internal query structure which may be treated as an identifier of the query as used by the library, and may be used as an argument for **dns_cancel**() routine. In case of error, NULL will be returned, and context error status (available using *dns_status*() routine) will be set to corresponding error code, which in this case may be DNS_E_BADQUERY if the *name* of *dn* is invalid, DNS_E_NOMEM if there's no memory available to allocate query structure, or DNS_E_TEMPFAIL if an internal error occured.

void ****dns_resolve_dn**(*ctx*,
const unsigned char ***dn*, *qcls*, *qtyp*, *flags*, *parse*);
void ****dns_resolve_p**(*ctx*,
const char ***name*, *qcls*, *qtyp*, *flags*, *parse*)
enum dns_class *qcls*;
enum dns_type *qtyp*;
int *flags*;
dns_parse_fn ***parse*;

syncronous interface. The routines perform all the steps necessary to resolve the given query and return the result. If there's no positive result for any reason, all the routines return NULL, and set context error status (available using **dns_status**() routine) to indicate the error code. If the query was successeful, context status code will contain either the length of the raw DNS reply packet if *parse* argument was NULL (in which case the return value is pointer to the reply DNS packet), or 0

(in which case the return value is the result of *parse* routine). If the query successeful (return value is not NULL), the memory returned was dynamically allocated by the library and should be free()d by application after use.

void *__dns_resolve__(*ctx*, struct dns_query **q*)
wait for the given query *q*, as returned by one of **dns_submit**_*XXX*() routines, for completion, and return the result. The callback routine will not be called for this query. After completion, the query identifier *q* is not valid. Both **dns_resolve_dn**() and **dns_resolve_p**() are just wrappers around corresponding submit routines and this **dns_resolve**() routine.

void **dns_cancel**(*ctx*, struct dns_query **q*)
cancel an active query *q*, without calling a callback routine. After completion, the query identifier *q* is not valid.

TYPE-SPECIFIC QUERIES

In addition to the generic low-level query interface, the library provides a set of routines to perform specific queries in a type-safe manner, as well as parsers for several well-known resource record types. The library implements high-level interface for A, AAAA, PTR, MX and TXT records and DNSBL and RHSBL functionality. These routines returns specific types as result of a query, instead of raw DNS packets. The following types and routines are available.

```
struct dns_rr_null {
  char *dnsn_qname;     /* original query name */
  char *dnsn_cname;     /* canonical name */
  unsigned dnsn_ttl;    /* Time-To-Live (TTL) value */
  int dnsn_nrr;         /* number of records in the set */
};
```

NULL RR set, used as a base for all other RR type structures. Every RR structure as used by the library have four standard fields as in struct **dns_rr_null**.

IN A Queries

```
struct dns_rr_a4 {       /* IN A RRset */
  char *dnsa4_qname;     /* original query name */
  char *dnsa4_cname;     /* canonical name */
```

```
  unsigned dnsa4_ttl;    /* Time-To-Live (TTL) value */
  int dnsa4_nrr;         /* number of addresses in the set */
  struct in_addr dnsa4_addr[]; /* array of addresses */
};
typedef void
  dns_query_a4_fn(ctx, struct dns_rr_a4 *result, data)
dns_parse_fn dns_parse_a4;
struct dns_query *
dns_submit_a4(ctx, const char *name, int flags,
   dns_query_a4_fn *cbck, data);
struct dns_rr_a4 *
dns_resolve_a4(ctx, const char *name, int flags);
```

The **dns_rr_a4** structure holds a result of an **IN A** query, which is an array of IPv4 addresses. Callback routine for IN A queries expected to be of type **dns_query_a4_fn**, which expects pointer to **dns_rr_a4** structure as query result instead of raw DNS packet. The **dns_parse_a4**() is used to convert raw DNS reply packet into **dns_rr_a4** structure (it is used internally and may be used directly too with generic query interface). Routines **dns_submit_a4**() and **dns_resolve_a4**() are used to perform A IN queries in a type-safe manner. The *name* parameter is the domain name in question, and *flags* is query flags bitmask, with one bit, DNS_NOSRCH, of practical interest (if the *name* is absolute, that is, it ends up with a dot, DNS_NOSRCH flag will be set automatically).

IN AAAA Queries

```
struct dns_rr_a6 {       /* IN AAAA RRset */
  char *dnsa6_qname;     /* original query name */
  char *dnsa6_cname;     /* canonical name */
  unsigned dnsa6_ttl;    /* Time-To-Live (TTL) value */
  int dnsa6_nrr;         /* number of addresses in the set */
  struct in6_addr dnsa6_addr[]; /* array of addresses */
};
typedef void
  dns_query_a6_fn(ctx, struct dns_rr_a6 *result, data)
dns_parse_fn dns_parse_a6;
struct dns_query *
dns_submit_a6(ctx, const char *name, int flags,
   dns_query_a6_fn *cbck, data);
struct dns_rr_a6 *
dns_resolve_a6(ctx, const char *name, int flags);
```

The **dns_rr_a6** structure holds a result of an **IN AAAA** query, which is an array of IPv6 addresses. Callback routine for IN AAAA queries expected to be of type **dns_query_a6_fn**, which expects pointer to **dns_rr_a6** structure as query result instead of raw DNS packet. The **dns_parse_a6**() is used to convert raw DNS reply packet into **dns_rr_a6** structure (it is used internally and may be used directly too with generic query interface). Routines **dns_submit_a6**() and **dns_resolve_a6**() are used to perform AAAA IN queries in a type-safe manner. The *name* parameter is the domain name in question, and *flags* is query flags bitmask, with one bit, DNS_NOSRCH, of practical interest (if the *name* is absolute, that is, it ends up with a dot, DNS_NOSRCH flag will be set automatically).

IN PTR Queries

```
struct dns_rr_ptr {       /* IN PTR RRset */
  char *dnsptr_qname;     /* original query name */
  char *dnsptr_cname;     /* canonical name */
  unsigned dnsptr_ttl;    /* Time-To-Live (TTL) value */
  int dnsptr_nrr;         /* number of domain name pointers */
  char *dnsptr_ptr[];     /* array of domain name pointers */
};
typedef void
  dns_query_ptr_fn(ctx, struct dns_rr_ptr *result, data)
dns_parse_fn dns_parse_ptr;
struct dns_query *
dns_submit_a4ptr(ctx, const struct in_addr *addr,
   dns_query_ptr_fn *cbck, data);
struct dns_rr_ptr *
dns_resolve_a4ptr(ctx, const struct in_addr *addr);
struct dns_query *
dns_submit_a6ptr(ctx, const struct in6_addr *addr,
   dns_query_ptr_fn *cbck, data);
struct dns_rr_ptr *
dns_resolve_a6ptr(ctx, const struct in6_addr *addr);
```

The **dns_rr_ptr** structure holds a result of an IN PTR query, which is an array of domain name pointers for a given IPv4 or IPv6 address. Callback routine for IN PTR queries expected to be of type **dns_query_ptr_fn**, which expects pointer to **dns_rr_ptr** structure as query result instead of raw DNS packet. The **dns_parse_ptr**() is used to convert raw DNS reply packet into **dns_rr_ptr** structure (it is used internally and may be used directly too with generic query interface). Routines **dns_submit_a4ptr**() and **dns_resolve_a4ptr**() are used to perform IN PTR queries for IPv4 addresses in a type-safe manner. Routines **dns_submit_a6ptr**() and **dns_resolve_a6ptr**() are used to perform IN PTR queries for IPv6 addresses.

IN MX Queries

```
struct dns_mx {         /* single MX record */
  int priority;         /* priority value of this MX */
  char *name;           /* domain name of this MX */
};
struct dns_rr_mx {      /* IN MX RRset */
  char *dnsmx_qname;    /* original query name */
  char *dnsmx_cname;    /* canonical name */
  unsigned dnsmx_ttl;   /* Time-To-Live (TTL) value */
  int dnsmx_nrr;        /* number of mail exchangers in the set */
  struct dns_mx dnsmx_mx[]; /* array of mail exchangers */
};
typedef void
  dns_query_mx_fn(ctx, struct dns_rr_mx *result, data)
dns_parse_fn dns_parse_mx;
struct dns_query *
dns_submit_mx(ctx, const char *name, int flags,
   dns_query_mx_fn *cbck, data);
struct dns_rr_mx *
dns_resolve_mx(ctx, const char *name, int flags);
```

The **dns_rr_mx** structure holds a result of an IN MX query, which is an array of mail exchangers for a given domain. Callback routine for IN MX queries expected to be of type **dns_query_mx_fn**, which expects pointer to **dns_rr_mx** structure as query result instead of raw DNS packet. The **dns_parse_mx**() is used to convert raw DNS reply packet into **dns_rr_mx** structure (it is used internally and may be used directly too with generic query interface). Routines **dns_submit_mx**() and **dns_resolve_mx**() are used to perform IN MX queries in a type-safe manner. The *name* parameter is the domain name in question, and *flags* is query flags bitmask, with one bit, DNS_NOSRCH, of practical interest (if the *name* is absolute, that is, it ends up with a dot, DNS_NOSRCH flag will be set automatically).

TXT Queries

```
struct dns_txt {        /* single TXT record */
  int len;              /* length of the text */
  unsigned char *txt;   /* pointer to the text */
};
struct dns_rr_txt {     /* TXT RRset */
  char *dnstxt_qname;   /* original query name */
  char *dnstxt_cname;   /* canonical name */
```

```
  unsigned dnstxt_ttl;    /* Time-To-Live (TTL) value */
  int dnstxt_nrr;         /* number of text records in the set */
  struct dns_txt dnstxt_txt[]; /* array of TXT records */
};
typedef void
  dns_query_txt_fn(ctx, struct dns_rr_txt *result, data)
dns_parse_fn dns_parse_txt;
struct dns_query *
dns_submit_txt(ctx, const char *name, enum dns_class qcls,
   int flags, dns_query_txt_fn *cbck, data);
struct dns_rr_txt *
dns_resolve_txt(ctx, const char *name,
          enum dns_class qcls, int flags);
```

The **dns_rr_txt** structure holds a result of a TXT query, which is an array of text records for a given domain name. Callback routine for TXT queries expected to be of type **dns_query_txt_fn**, which expects pointer to **dns_rr_txt** structure as query result instead of raw DNS packet. The **dns_parse_txt**() is used to convert raw DNS reply packet into **dns_rr_txt** structure (it is used internally and may be used directly too with generic query interface). Routines **dns_submit_txt**() and **dns_resolve_txt**() are used to perform IN MX queries in a type-safe manner. The *name* parameter is the domain name in question, and *flags* is query flags bitmask, with one bit, DNS_NOSRCH, of practical interest (if the *name* is absolute, that is, it ends up with a dot, DNS_NOSRCH flag will be set automatically). Note that each TXT string is represented by **struct dns_txt**, while zero-terminated (and the len field of the structure does not include the terminator), may contain embedded null characters -- content of TXT records is not interpreted by the library in any way.

SRV Queries

```
struct dns_srv {        /* single SRV record */
  int priority;         /* priority of the record */
  int weight;           /* weight of the record */
  int port;             /* the port number to connect to */
  char *name;           /* target host name */
};
struct dns_rr_srv {     /* SRV RRset */
  char *dnssrv_qname;   /* original query name */
  char *dnssrv_cname;   /* canonical name */
  unsigned dnssrv_ttl;  /* Time-To-Live (TTL) value */
  int dnssrv_nrr;       /* number of text records in the set */
  struct dns_srv dnssrv_srv[]; /* array of SRV records */
```

```
};
typedef void
  dns_query_srv_fn(ctx, struct dns_rr_srv *result, data)
dns_parse_fn dns_parse_srv;
struct dns_query *
dns_submit_srv(ctx, const char *name, const char *service, const char *protocol,
  int flags, dns_query_txt_fn *cbck, data);
struct dns_rr_srv *
dns_resolve_srv(ctx, const char *name, const char *service, const char *protocol,
        int flags);
```

The **dns_rr_srv** structure holds a result of an IN SRV (rfc2782) query, which is an array of servers (together with port numbers) which are performing operations for a given *service* using given *protocol* on a target domain *name*. Callback routine for IN SRV queries expected to be of type **dns_query_srv_fn**, which expects pointer to **dns_rr_srv** structure as query result instead of raw DNS packet. The **dns_parse_srv**() is used to convert raw DNS reply packet into **dns_rr_srv** structure (it is used internally and may be used directly too with generic query interface). Routines **dns_submit_srv**() and **dns_resolve_srv**() are used to perform IN SRV queries in a type-safe manner. The *name* parameter is the domain name in question, *service* and protocl specifies the service and the protocol in question (the library will construct query DN according to rfc2782 rules) and may be NULL (in this case the library assumes *name* parameter holds the complete SRV query), and *flags* is query flags bitmask, with one bit, DNS_NOSRCH, of practical interest (if the *name* is absolute, that is, it ends up with a dot, DNS_NOSRCH flag will be set automatically).

NAPTR Queries

```
struct dns_naptr {       /* single NAPTR record */
  int order;             /* record order */
  int preference;        /* preference of this record */
  char *flags;           /* application-specific flags */
  char *service;         /* service parameter */
  char *regexp;          /* substitutional regular expression */
  char *replacement;     /* replacement string */
};
struct dns_rr_naptr {    /* NAPTR RRset */
  char *dnsnaptr_qname;  /* original query name */
  char *dnsnaptr_cname;  /* canonical name */
  unsigned dnsnaptr_ttl; /* Time-To-Live (TTL) value */
  int dnsnaptr_nrr;      /* number of text records in the set */
  struct dns_naptr dnsnaptr_naptr[]; /* array of NAPTR records */
```

};
typedef void
dns_query_naptr_fn(*ctx*, struct dns_rr_naptr ***result*, *data*)
dns_parse_fn **dns_parse_naptr;**
struct dns_query *
dns_submit_naptr(*ctx*, const char ***name*, int *flags*,
dns_query_txt_fn ***cbck*, *data*);
struct dns_rr_naptr *
dns_resolve_naptr(*ctx*, const char ***name*, int *flags*);

The **dns_rr_naptr** structure holds a result of an IN NAPTR (rfc3403) query. Callback routine for IN NAPTR queries expected to be of type **dns_query_naptr_fn**, expects pointer to **dns_rr_naptr** structure as query result instead of raw DNS packet. The **dns_parse_naptr**() is used to convert raw DNS reply packet into **dns_rr_naptr** structure (it is used internally and may be used directly too with generic query interface). Routines **dns_submit_naptr**() and **dns_resolve_naptr**() are used to perform IN NAPTR queries in a type-safe manner. The *name* parameter is the domain name in question, and *flags* is query flags bitmask, with one bit, DNS_NOSRCH, of practical interest (if the *name* is absolute, that is, it ends up with a dot, DNS_NOSRCH flag will be set automatically).

DNSBL Interface

A DNS-based blocklists, or a DNSBLs, are in wide use nowadays, especially to protect mailservers from spammers. The library provides DNSBL interface, a set of routines to perform queries against DNSBLs. Routines accepts an IP address (IPv4 and IPv6 are both supported) and a base DNSBL zone as query parameters, and returns either **dns_rr_a4** or **dns_rr_txt** structure. Note that IPv6 interface return IPv4 RRset.

struct dns_query *
dns_submit_a4dnsbl(*ctx*,
const struct in_addr ***addr*, const char ***dnsbl*,
dns_query_a4_fn ***cbck*, void ***data*);
struct dns_query *
dns_submit_a4dnsbl_txt(*ctx*,
const struct in_addr ***addr*, const char ***dnsbl*,
dns_query_txt_fn ***cbck*, void ***data*);
struct dns_query *
dns_submit_a6dnsbl(*ctx*,
const struct in6_addr ***addr*, const char ***dnsbl*,
dns_query_a4_fn ***cbck*, void ***data*);
struct dns_query *

dns_submit_a6dnsbl_txt(*ctx*,
const struct in6_addr **addr*, const char **dnsbl*,
dns_query_txt_fn **cbck*, void **data*);
struct dns_rr_a4 ***dns_resolve_a4dnsbl**(*ctx*,
const struct in_addr **addr*, const char **dnsbl*)
struct dns_rr_txt ***dns_resolve_a4dnsbl_txt**(*ctx*,
const struct in_addr **addr*, const char **dnsbl*)
struct dns_rr_a4 ***dns_resolve_a6dnsbl**(*ctx*,
const struct in6_addr **addr*, const char **dnsbl*)
struct dns_rr_txt ***dns_resolve_a6dnsbl_txt**(*ctx*,
const struct in6_addr **addr*, const char **dnsbl*)
Perform (submit or resolve) a DNSBL query for the given *dnsbl* domain and an IP *addr* in question, requesting either A or TXT records.

RHSBL Interface

RHSBL is similar to DNSBL, but instead of an IP address, the parameter is a domain name.

struct dns_query *
dns_submit_rhsbl(*ctx*, const char **name*, const char **rhsbl*,
dns_query_a4_fn **cbck*, void **data*);
struct dns_query *
dns_submit_rhsbl_txt(*ctx*, const char **name*, const char **rhsbl*,
dns_query_txt_fn **cbck*, void **data*);
struct dns_rr_a4 *
dns_resolve_rhsbl(*ctx*, const char **name*, const char **rhsbl*);
struct dns_rr_txt *
dns_resolve_rhsbl_txt(*ctx*, const char **name*, const char **rhsbl*);
Perform (submit or resolve) a RHSBL query for the given *rhsbl* domain and *name* in question, requesting either A or TXT records.

LOW-LEVEL INTERFACE

Domain Names (DNs)

A DN is a series of domain name labels each starts with length byte, followed by empty label (label with zero length). The following routines to work with DNs are provided.

unsigned **dns_dnlen**(const unsigned char **dn*)

return length of the domain name *dn*, including the terminating label.

unsigned **dns_dnlabels**(const unsigned char **dn*)
return number of non-zero labels in domain name *dn*.

unsigned **dns_dnequal**(*dn1*, *dn2*)
const unsigned char **dn1*, **dn2*;
test whenever the two domain names, *dn1* and *dn2*, are equal (case-insensitive). Return domain name length if equal or 0 if not.

unsigned **dns_dntodn**(*sdn*, *ddn*, *dnsiz*)
const unsigned char **sdn*;
unsigned char **ddn*;
unsigned *dnsiz*;
copies the source domain name *sdn* to destination buffer *ddn* of size *dnsiz*. Return domain name length or 0 if *ddn* is too small.

int **dns_ptodn**(*name*, *namelen*, *dn*, *dnsiz*, *isabs*)
int **dns_sptodn**(*name*, *dn*, *dnsiz*)
const char ***name*; unsigned *namelen*;
unsigned char ***dn*; unsigned *dnsiz*;
int ***isabs*;
convert asciiz name *name* of length *namelen* to DN format, placing result into buffer *dn* of size *dnsiz*. Return length of the DN if successeful, 0 if the *dn* buffer supplied is too small, or negative value if *name* is invalid. If *isabs* is non-NULL and conversion was successeful, ***isabs* will be set to either 1 or 0 depending whenever *name* was absolute (i.e. ending with a dot) or not. Name length, *namelength*, may be zero, in which case strlen(*name*) will be used. Second form, **dns_sptodn**(), is a simplified form of **dns_ptodn**(), equivalent to **dns_ptodn**(*name*, 0, *dn*, *dnlen*, 0).

extern const unsigned char **dns_inaddr_arpa_dn**[]
int **dns_a4todn**(const struct in_addr ***addr*, const unsigned char ***tdn*,
unsigned char ***dn*, unsigned *dnsiz*)
int **dns_a4ptodn**(const struct in_addr ***addr*, const char ***tname*,
unsigned char ***dn*, unsigned *dnsiz*)

extern const unsigned char **dns_ip6_arpa_dn**[]
int **dns_a6todn**(const struct in6_addr *_addr_, const unsigned char *_tdn_,
unsigned char *_dn_, unsigned _dnsiz_)
int **dns_a6ptodn**(const struct in6_addr *_addr_, const char *_tname_,
unsigned char *_dn_, unsigned _dnsiz_)
several variants of routines to convert IPv4 and IPv6 address _addr_ into reverseDNS-like domain name in DN format, storing result in _dn_ of size _dnsiz_. _tdn_ (or _tname_) is the base zone name, like in-addr.arpa for IPv4 or in6.arpa for IPv6. If _tdn_ (or _tname_) is NULL, **dns_inaddr_arpa_dn** (or **dns_ip6_arpa_dn**) will be used. The routines may be used to construct a DN for a DNSBL lookup for example. All routines return length of the resulting DN on success, -1 if resulting DN is invalid, or 0 if the _dn_ buffer (_dnsiz_) is too small. To hold standard rDNS DN, a buffer of size **DNS_A4RSIZE** (30 bytes) for IPv4 address, or **DNS_A6RSIZE** (74 bytes) for IPv6 address, is sufficient.

int **dns_dntop**(_dn_, _name_, _namesiz_)
const unsigned char *_dn_;
const char *_name_; unsigned _namesiz_;
convert domain name _dn_ in DN format to asciiz string, placing result into _name_ buffer of size _namesiz_. Maximum length of asciiz representation of domain name is **DNS_MAXNAME** (1024) bytes. Root domain is represented as empty string. Return length of the resulting name (including terminating character, i.e. strlen(name)+1) on success, 0 if the _name_ buffer is too small, or negative value if _dn_ is invalid (last case should never happen since all routines in this library which produce domain names ensure the DNs generated are valid).

const char *__dns_dntosp__(const unsigned char *_dn_)
convert domain name _dn_ in DN format to asciiz string using static buffer. Return the resulting asciiz string on success or NULL on failure. Note since this routine uses static buffer, it is not thread-safe.

unsigned **dns_dntop_size**(const unsigned char *_dn_)
return the buffer size needed to convert the _dn_ domain name in DN format to asciiz string, for **dns_dntop**(). The routine return either the size of buffer required, including the trailing zero byte, or 0 if _dn_ is invalid.

Working with DNS Packets

The following routines are provided to encode and decode DNS on-wire packets. This is low-level

interface.

DNS response codes (returned by **dns_rcode**() routine) are defined as constants prefixed with **DNS_R_**. See udns.h header file for the complete list. In particular, constants **DNS_R_NOERROR** (0), **DNS_R_SERVFAIL**, **DNS_R_NXDOMAIN** may be of interest to an application.

unsigned **dns_get16**(const unsigned char *_p_)
unsigned **dns_get32**(const unsigned char *_p_)
helper routines, convert 16-bit or 32-bit integer in on-wire format pointed to by _p_ to unsigned.

unsigned char *__dns_put16__(unsigned char *_d_, unsigned _n_)
unsigned char *__dns_put32__(unsigned char *_d_, unsigned _n_)
helper routine, convert unsigned 16-bit or 32-bit integer _n_ to on-wire format to buffer pointed to by _d_, return _d_+2 or _d_+4.

DNS_HSIZE (12)
defines size of DNS header. Data section in the DNS packet immediately follows the header. In the header, there are query identifier (id), various flags and codes, and number of resource records in various data sections. See udns.h header file for complete list of DNS header definitions.

unsigned **dns_qid**(const unsigned char *_pkt_)
int **dns_rd**(const unsigned char *_pkt_)
int **dns_tc**(const unsigned char *_pkt_)
int **dns_aa**(const unsigned char *_pkt_)
int **dns_qr**(const unsigned char *_pkt_)
int **dns_ra**(const unsigned char *_pkt_)
unsigned **dns_opcode**(const unsigned char *_pkt_)
unsigned **dns_rcode**(const unsigned char *_pkt_)
unsigned **dns_numqd**(const unsigned char *_pkt_)
unsigned **dns_numan**(const unsigned char *_pkt_)
unsigned **dns_numns**(const unsigned char *_pkt_)
unsigned **dns_numar**(const unsigned char *_pkt_)
const unsigned char *__dns_payload__(const unsigned char *_pkt_)
return various parts from the DNS packet header _pkt_: query identifier (qid), recursion desired (rd) flag, truncation occured (tc) flag, authoritative answer (aa) flag, query response (qr) flag, recursion

available (ra) flag, operation code (opcode), result code (rcode), number of entries in question section (numqd), number of answers (numan), number of authority records (numns), number of additional records (numar), and the pointer to the packet data (payload).

int **dns_getdn**(*pkt*, *curp*, *pkte*, *dn*, *dnsiz*)
const unsigned char *__dns_skipdn__(*cur*, *pkte*)
const unsigned char **pkt*, **pkte*, ***curp*, **cur*;
unsigned char **dn*; unsigned *dnsiz*;

dns_getdn() extract DN from DNS packet *pkt* which ends before *pkte* starting at position **curp* into buffer pointed to by *dn* of size *dnsiz*. Upon successeful completion, **curp* will point to the next byte in the packet after the extracted domain name. It return positive number (length of the DN if *dn*) upon successeful completion, negative value on error (when the packet contains invalid data), or zero if the *dnsiz* is too small (maximum length of a domain name is **DNS_MAXDN**).

dns_skipdn() return pointer to the next byte in DNS packet which ends up before *pkte* after a domain name which starts at the *cur* byte, or NULL if the packet is invalid. **dns_skipdn**() is more or less equivalent to what **dns_getdn**() does, except it does not actually extract the domain name in question, and uses simpler interface.

```
struct dns_rr {
  unsigned char dnsrr_dn[DNS_MAXDN]; /* the RR DN name */
  enum dns_class dnsrr_cls;         /* class of the RR */
  enum dns_type  dnsrr_typ;         /* type of the RR */
  unsigned dnsrr_ttl;               /* TTL value */
  unsigned dnsrr_dsz;               /* size of data in bytes */
  const unsigned char *dnsrr_dptr;  /* pointer to the first data byte */
  const unsigned char *dnsrr_dend;  /* next byte after RR */
};
```

The **dns_rr** structure is used to hold information about single DNS Resource Record (RR) in an easy to use form.

```
struct dns_parse {
  const unsigned char *dnsp_pkt; /* pointer to the packet being parsed */
  const unsigned char *dnsp_end; /* end of the packet pointer */
  const unsigned char *dnsp_cur; /* current packet positionn */
  const unsigned char *dnsp_ans; /* pointer to the answer section */
  int dnsp_rrl;                 /* number of RRs left */
  int dnsp_nrr;                 /* number of relevant RRs seen so far */
```

```
  unsigned dnsp_ttl;          /* TTL value so far */
  const unsigned char *dnsp_qdn; /* the domain of interest or NULL */
  enum dns_class dnsp_qcls;     /* class of interest or 0 for any */
  enum dns_type  dnsp_qtyp;     /* type of interest or 0 for any */
  unsigned char dnsp_dnbuf[DNS_MAXDN]; /* domain name buffer */
};
```

The **dns_parse** structure is used to parse DNS reply packet. It holds information about the packet being parsed (dnsp_pkt, dnsp_end and dnsp_cur fields), number of RRs in the current section left to do, and the information about specific RR which we're looking for (dnsp_qdn, dnsp_qcls and dnsp_qtyp fields).

int **dns_initparse**(struct dns_parse *p,
const unsigned char *qdn,
const unsigned char *pkt,
const unsigned char *cur,
const unsigned char *end)

initializes the RR parsing structure *p*. Arguments *pkt*, *cur* and *end* should describe the received packet: *pkt* is the start of the packet, *end* points to the next byte after the end of the packet, and *cur* points past the query DN in query section (to query class+type information). And *qdn* points to the query DN. This is the arguments passed to **dns_parse_fn**() routine. **dns_initparse**() initializes **dnsp_pkt**, **dnsp_end** and **dnsp_qdn** fields to the corresponding arguments, extracts and initializes **dnsp_qcls** and **dnsp_qtyp** fields to the values found at *cur* pointer, initializes **dnsp_cur** and **dnsp_ans** fields to be *cur*+4 (to the start of answer section), and initializes **dnsp_rrl** field to be number of entries in answer section. **dnsp_ttl** will be set to max TTL value, 0xffffffff, and **dnsp_nrr** to 0.

int **dns_nextrr**(struct dns_parse *p, struct dns_rr *rr);

searches for next RR in the packet based on the criteria provided in the *p* structure, filling in the *rr* structure and advancing *p*->**dnsp_cur** to the next RR in the packet. RR selection is based on dnsp_qdn, dnsp_qcls and dnsp_qtyp fields in the dns_parse structure. Any (or all) of the 3 fields may be 0, which means any actual value from the packet is acceptable. In case the field isn't 0 (or NULL for dnsp_qdn), only RRs with corresponding characteristics are acceptable. Additionally, when dnsp_qdn is non-NULL, **dns_nextrr**() performs automatic CNAME expansion. Routine will return positive value on success, 0 in case it reached the end of current section in the packet (*p*->**dnsp_rrl** is zero), or negative value if next RR can not be decoded (packet format is invalid). The routine updates *p*->**dnsp_qdn** automatically when this field is non-NULL and it encounters appropriate CNAME RRs (saving CNAME target in *p*->**dnsp_dnbuf**), so after end of the process, *p*->**dnsp_qdn** will point to canonical name of the domain in question. The routine updates

p->**dnsp_ttl** value to be the minimum TTL of all RRs found.

void **dns_rewind**(struct dns_parse **p*, const unsigned char **qdn*)
this routine "rewinds" the packet parse state structure to be at the same state as after a call to **dns_initparse**(), i.e. reposition the parse structure *p* to the start of answer section and initialize *p*->**dnsp_rrl** to the number of entries in answer section.

int **dns_stdrr_size**(const struct dns_parse **p*);
return size to hold standard RRset structure information, as shown in **dns_rr_null** structure (for the query and canonical names). Used to calculate amount of memory to allocate for common part of type-specific RR structures in parsing routines.

void ***dns_stdrr_finish**(struct dns_rr_null **ret*, char **cp*,
const struct dns_parse **p*);
initializes standard RRset fields in *ret* structure using buffer pointed to by *cp*, which should have at least as many bytes as **dns_stdrr_size**(*p*) returned. Used to finalize common part of type-specific RR structures in parsing routines.

See library source for usage examples of all the above low-level routines, especially source of the parsing routines.

Auxilary Routines

int **dns_pton**(int *af*, const char **src*, void **dst*);
prives functionality similar to standard **inet_pton**() routine, to convert printable representation of an IP address of family *af* (either **AF_INET** or **AF_INET6**) pointed to by *src* into binary form suitable for socket addresses and transmission over network, in buffer pointed to by *dst*. The destination buffer should be of size 4 for **AF_INET** family or 16 for **AF_INET6**. The return value is positive on success, 0 if *src* is not a valid text representation of an address of family *af*, or negative if the given address family is not supported.

const char ***dns_ntop**(int *af*, const void **src*,
char **dst*, int *dstsize*)
prives functionality similar to standard **inet_ntop**() routine, to convert binary representation of an IP address of family *af* (either **AF_INET** or **AF_INET6**) pointed to by *src* (either 4 or 16 bytes)

into printable form in buffer in buffer pointed to by *dst* of size *dstsize*. The destination buffer should be at least of size 16 bytes for **AF_INET** family or 46 bytes for **AF_INET6**. The return value is either *dst*, or NULL pointer if *dstsize* is too small to hold this address or if the given address family is not supported.

AUTHOR

The **udns** library has been written by Michael Tokarev, mjt+udns@tls.msk.ru.

VERSION

This manual page corresponds to udns version 0.4, released Jan-2014.