NAME

snmp_client, **snmp_client_init**, **snmp_client_set_host**, **snmp_client_set_port**, **snmp_send_cb_f**, **snmp_timeout_cb_f**, **snmp_timeout_start_f**, **snmp_timeout_stop_f**, **snmp_open**, **snmp_close**, **snmp_pdu_create**, **snmp_add_binding**, **snmp_pdu_check**, **snmp_pdu_send**, **snmp_oid_append**, **snmp_parse_server**, **snmp_receive**, **snmp_table_cb_f**, **snmp_table_fetch**, **snmp_table_fetch_async**, **snmp_dialog**, **snmp_discover_engine** - SNMP client library

LIBRARY

Begemot SNMP library (libbsnmp, -lbsnmp)

SYNOPSIS

#include <asn1.h>
#include <snmp.h>
#include <snmpclient.h>

typedef void
(*snmp_send_cb_f)(*struct snmp_pdu *req, struct snmp_pdu *resp, void *uarg*);

typedef void
(*snmp_timeout_cb_f)(*void *uarg*);

*typedef void **
(*snmp_timeout_start_f)(*struct timeval *timeout, snmp_timeout_cb_f callback, void *uarg*);

typedef void
(*snmp_timeout_stop_f)(*void *timeout_id*);

extern struct snmp_client snmp_client;

void
snmp_client_init(*struct snmp_client *client*);

int
snmp_client_set_host(*struct snmp_client *client, const char *host*);

int
snmp_client_set_port(*struct snmp_client *client, const char *port*);

int
snmp_open(*const char *host, const char *port, const char *read_community,*

*const char *write_community*);

void
snmp_close(*void*);

void
snmp_pdu_create(*struct snmp_pdu *pdu*, *u_int op*);

int
snmp_add_binding(*struct snmp_pdu *pdu*, ...);

int
snmp_pdu_check(*const struct snmp_pdu *req*, *const struct snmp_pdu *resp*);

int32_t
snmp_pdu_send(*struct snmp_pdu *pdu*, *snmp_send_cb_f func*, *void *uarg*);

int
snmp_oid_append(*struct asn_oid *oid*, *const char *fmt*, ...);

int
snmp_parse_server(*struct snmp_client *sc*, *const char *str*);

int
snmp_receive(*int blocking*);

typedef void
(*snmp_table_cb_f)(*void *list*, *void *arg*, *int res*);

int
snmp_table_fetch(*const struct snmp_table *descr*, *void *list*);

int
snmp_table_fetch_async(*const struct snmp_table *descr*, *void *list*, *snmp_table_cb_f callback*, *void *uarg*);

int
snmp_dialog(*struct snmp_pdu *req*, *struct snmp_pdu *resp*);

int

snmp_discover_engine(*void*);

DESCRIPTION

The SNMP library contains routines to easily build SNMP client applications that use SNMP versions 1, 2 or 3. Most of the routines use a *struct snmp_client*:

```
struct snmp_client {
        enum snmp_version       version;
        int                     trans;      /* which transport to use */

        /* these two are read-only for the application */
        char                    *cport;     /* port number as string */
        char                    *chost;     /* host name or IP address as string */

        char                    read_community[SNMP_COMMUNITY_MAXLEN + 1];
        char                    write_community[SNMP_COMMUNITY_MAXLEN + 1];

        /* SNMPv3 specific fields */
        int32_t                 identifier;
        int32_t                 security_model;
        struct snmp_engine engine;
        struct snmp_user    user;

        /* SNMPv3 Access control - VACM*/
        uint32_t            clen;
        uint8_t                 cengine[SNMP_ENGINE_ID_SIZ];
        char                    cname[SNMP_CONTEXT_NAME_SIZ];

        struct timeval          timeout;
        u_int                   retries;

        int                     dump_pdus;

        size_t                  txbuflen;
        size_t                  rxbuflen;

        int                     fd;

        int32_t                 next_reqid;
        int32_t                 max_reqid;
```

```
        int32_t                 min_reqid;

        char                    error[SNMP_STRERROR_LEN];

        snmp_timeout_start_f    timeout_start;
        snmp_timeout_stop_f     timeout_stop;

        char                    local_path[sizeof(SNMP_LOCAL_PATH)];
};
```

The fields of this structure are described below.

version This is the version of SNMP to use. See bsnmplib(3) for applicable values. The default version is SNMP_V2c.

trans If this is SNMP_TRANS_LOC_DGRAM a local datagram socket is used. If it is SNMP_TRANS_LOC_STREAM a local stream socket is used. For SNMP_TRANS_UDP a UDPv4 socket and for SNMP_TRANS_UDP6 a UDPv6 socket is created. It uses the *chost* field as the path to the server's socket for local sockets.

cport The SNMP agent's UDP port number. This may be a symbolic port number (from */etc/services*) or a numeric port number. If this field is NULL (the default) the standard SNMP port is used. This field should not be changed directly but rather by calling **snmp_client_set_port**().

chost The SNMP agent's host name, IP address or UNIX domain socket path name. If this is NULL (the default) localhost is assumed. This field should not be changed directly but rather through calling **snmp_client_set_host**().

read_community
This is the community name to be used for all requests except SET requests. The default is 'public'.

write_community
The community name to be used for SET requests. The default is 'private'.

identifier The message identifier value to be used with SNMPv3 PDUs. Incremented with each transmitted PDU.

security_model

The security model to be used with SNMPv3 PDUs. Currently only User-Based Security model specified by RFC 3414 (value 3) is supported.

engine The authoritive SNMP engine parameters to be used with SNMPv3 PDUs.

user The USM SNMP user credentials to be used with SNMPv3 PDUs.

clen The length of the context engine id to be used with SNMPv3 PDUs.

cengine The context engine id to be used with SNMPv3 PDUs. Default is empty.

cname The context name to be used with SNMPv3 PDUs. Default is ‘’.

timeout The maximum time to wait for responses to requests. If the time elapses, the request is resent up to *retries* times. The default is 3 seconds.

retries Number of times a request PDU is to be resent. If set to 0, the request is sent only once. The default is 3 retransmissions.

dump_pdus If set to a non-zero value all received and sent PDUs are dumped via snmp_pdu_dump(3). The default is not to dump PDUs.

txbuflen The encoding buffer size to be allocated for transmitted PDUs. The default is 10000 octets.

rxbuflen The decoding buffer size to be allocated for received PDUs. This is the size of the maximum PDU that can be received. The default is 10000 octets.

fd After calling **snmp_open**() this is the file socket file descriptor used for sending and receiving PDUs.

next_reqid The request id of the next PDU to send. Used internal by the library.

max_reqid The maximum request id to use for outgoing PDUs. The default is INT32_MAX.

min_reqid The minimum request id to use for outgoing PDUs. Request ids are allocated linearily starting at *min_reqid* up to *max_reqid*.

error If an error happens, this field is set to a printable string describing the error.

timeout_start This field must point to a function setting up a one shot timeout. After the timeout has elapsed, the given callback function must be called with the user argument. The **timeout_start**() function must return a *void* * identifying the timeout.

timeout_stop This field must be set to a function that stops a running timeout. The function will be called with the return value of the corresponding **timeout_start**() function.

local_path If in local socket mode, the name of the clients socket. Not needed by the application.

In the current implementation there is a global variable

extern struct snmp_client snmp_client;

that is used by all the library functions. The first call into the library must be a call to **snmp_client_init**() to initialize this global variable to the default values. After this call and before calling **snmp_open**() the fields of the variable may be modified by the user. The modification of the *chost* and *cport* fields should be done only via the functions **snmp_client_set_host**() and **snmp_client_set_port**().

The function **snmp_open**() creates a UDP or UNIX domain socket and connects it to the agent's IP address and port. If any of the arguments of the call is not NULL the corresponding field in the global *snmp_client* is set from the argument. Otherwise the values that are already in that variable are used. The function **snmp_close**() closes the socket, stops all timeouts and frees all dynamically allocated resources.

The next three functions are used to create request PDUs. The function **snmp_pdu_create**() initializes a PDU of type *op*. It does not allocate space for the PDU itself. This is the responsibility of the caller. **snmp_add_binding**() adds bindings to the PDU and returns the (zero based) index of the first new binding. The arguments are pairs of pointer to the OIDs and syntax constants, terminated by a NULL. The call

```
snmp_add_binding(&pdu,
    &oid1, SNMP_SYNTAX_INTEGER,
    &oid2, SNMP_SYNTAX_OCTETSTRING,
    NULL);
```

adds two new bindings to the PDU and returns the index of the first one. It is the responsibility of the caller to set the value part of the binding if necessary. The functions returns -1 if the maximum number of bindings is exhausted. The function **snmp_oid_append**() can be used to construct variable OIDs for requests. It takes a pointer to an *struct asn_oid* that is to be constructed, a format string, and a number of arguments the type of which depends on the format string. The format string is interpreted character by

character in the following way:

i This format expects an argument of type *asn_subid_t* and appends this as a single integer to the OID.

a This format expects an argument of type *struct in_addr* and appends to four parts of the IP address to the OID.

s This format expects an argument of type *const char* * and appends the length of the string (as computed by strlen(3)) and each of the characters in the string to the OID.

(*N*) This format expects no argument. *N* must be a decimal number and is stored into an internal variable *size*.

b This format expects an argument of type *const char* * and appends *size* characters from the string to the OID. The string may contain NUL characters.

c This format expects two arguments: one of type *size_t* and one of type *const u_char* *. The first argument gives the number of bytes to append to the OID from the string pointed to by the second argument.

The function **snmp_pdu_check**() may be used to check a response PDU. A number of checks are performed (error code, equal number of bindings, syntaxes and values for SET PDUs). The function returns +1 if everything is ok, 0 if a NOSUCHNAME or similar error was detected, -1 if the response PDU had fatal errors and -2 if *resp* is NULL (a timeout occurred).

The function **snmp_pdu_send**() encodes and sends the given PDU. It records the PDU together with the callback and user pointers in an internal list and arranges for retransmission if no response is received. When a response is received or the retransmission count is exceeded the callback *func* is called with the original request PDU, the response PDU and the user argument *uarg*. If the retransmit count is exceeded, *func* is called with the original request PDU, the response pointer set to NULL and the user argument *uarg*. The caller should not free the request PDU until the callback function is called. The callback function must free the request PDU and the response PDU (if not NULL).

The function **snmp_receive**() tries to receive a PDU. If the argument is zero, the function polls to see whether a packet is available, if the argument is non-zero, the function blocks until the next packet is received. The packet is delivered via the usual callback mechanism (non-response packets are silently dropped). The function returns 0, if a packet was received and successfully dispatched, -1 if an error occurred or no packet was available (in polling mode).

The next two functions are used to retrieve tables from SNMP agents. They use the following input structure, that describes the table:

```
struct snmp_table {
        struct asn_oid          table;
        struct asn_oid          last_change;
        u_int                   max_iter;
        size_t                  entry_size;
        u_int                   index_size;
        uint64_t        req_mask;

        struct snmp_table_entry {
            asn_subid_t         subid;
            enum snmp_syntax    syntax;
            off_t           offset;
        }                       entries[];
};
```

The fields of this structure have the following meaning:

table This is the base OID of the table.

last_change Some tables have a scalar variable of type TIMETICKS attached to them, that holds the time when the table was last changed. This OID should be the OID of this variable (without the .0 index). When the table is retrieved with multiple GET requests, and the variable changes between two request, the table fetch is restarted.

max_iter Maximum number of tries to fetch the table.

entry_size The table fetching routines return a list of structures one for each table row. This variable is the size of one structure and used to malloc(3) the structure.

index_size This is the number of index columns in the table.

req_mask This is a bit mask with a 1 for each table column that is required. Bit 0 corresponds to the first element (index 0) in the array *entries*, bit 1 to the second (index 1) and so on. SNMP tables may be sparse. For sparse columns the bit should not be set. If the bit for a given column is set and the column value cannot be retrieved for a given row, the table fetch is restarted assuming that the table is currently being modified by the agent. The bits for the index columns are ignored.

entries This is a variable sized array of column descriptors. This array is terminated by an element with syntax SNMP_SYNTAX_NULL. The first *index_size* elements describe all the index columns of the table, the rest are normal columns. If for the column at 'entries[N]' the expression 'req_mask & (1 << N)' yields true, the column is considered a required column. The fields of this the array elements have the following meaning:

subid This is the OID subid of the column. This is ignored for index entries. Index entries are decoded according to the *syntax* field.

syntax This is the syntax of the column or index. A syntax of SNMP_SYNTAX_NULL terminates the array.

offset This is the starting offset of the value of the column in the return structures. This field can be set with the ISO-C **offsetof**() macro.

Both table fetching functions return TAILQ (see queue(3)) of structures--one for each table row. These structures must start with a **TAILQ_ENTRY**() and a *uint64_t* and are allocated via malloc(3). The *list* argument of the table functions must point to a **TAILQ_HEAD**(). The *uint64_t* fields, usually called *found* is used to indicate which of the columns have been found for the given row. It is encoded like the *req_mask* field.

The function **snmp_table_fetch**() synchronously fetches the given table. If everything is ok 0 is returned. Otherwise the function returns -1 and sets an appropriate error string. The function **snmp_table_fetch_async**() fetches the tables asynchronously. If either the entire table is fetch, or an error occurs the callback function *callback* is called with the callers arguments *list* and *uarg* and a parameter that is either 0 if the table was fetched, or -1 if there was an error. The function itself returns -1 if it could not initialize fetching of the table.

The following table description is used to fetch the ATM interface table:

```
/*
 * ATM interface table
 */
struct atmif {
        TAILQ_ENTRY(atmif) link;
        uint64_t   found;
        int32_t            index;
        u_char             *ifname;
        size_t             ifnamelen;
        uint32_t   node_id;
```

```
        uint32_t  pcr;
        int32_t           media;
        uint32_t  vpi_bits;
        uint32_t  vci_bits;
        uint32_t  max_vpcs;
        uint32_t  max_vccs;
        u_char            *esi;
        size_t            esilen;
        int32_t           carrier;
};
TAILQ_HEAD(atmif_list, atmif);

/* list of all ATM interfaces */
struct atmif_list atmif_list;

static const struct snmp_table atmif_table = {
        OIDX_begemotAtmIfTable,
        OIDX_begemotAtmIfTableLastChange, 2,
        sizeof(struct atmif),
        1, 0x7ffULL,
        {
          { 0, SNMP_SYNTAX_INTEGER,
                offsetof(struct atmif, index) },
          { 1, SNMP_SYNTAX_OCTETSTRING,
                offsetof(struct atmif, ifname) },
          { 2, SNMP_SYNTAX_GAUGE,
                offsetof(struct atmif, node_id) },
          { 3, SNMP_SYNTAX_GAUGE,
                offsetof(struct atmif, pcr) },
          { 4, SNMP_SYNTAX_INTEGER,
                offsetof(struct atmif, media) },
          { 5, SNMP_SYNTAX_GAUGE,
                offsetof(struct atmif, vpi_bits) },
          { 6, SNMP_SYNTAX_GAUGE,
                offsetof(struct atmif, vci_bits) },
          { 7, SNMP_SYNTAX_GAUGE,
                offsetof(struct atmif, max_vpcs) },
          { 8, SNMP_SYNTAX_GAUGE,
                offsetof(struct atmif, max_vccs) },
          { 9, SNMP_SYNTAX_OCTETSTRING,
```

```
                        offsetof(struct atmif, esi) },
                { 10, SNMP_SYNTAX_INTEGER,
                        offsetof(struct atmif, carrier) },
        { 0, SNMP_SYNTAX_NULL, 0 }
            }
};

...
            if (snmp_table_fetch(&atmif_table, &atmif_list) != 0)
                        errx(1, "AtmIf table: %s", snmp_client.error);
...
```

The function **snmp_dialog**() is used to execute a synchonuous dialog with the agent. The request PDU *req* is sent and the function blocks until the response PDU is received. Note, that asynchonuous receives are handled (i.e. callback functions of other send calls or table fetches may be called while in the function). The response PDU is returned in *resp*. If no response could be received after all timeouts and retries, the function returns -1. If a response was received 0 is returned.

The function **snmp_discover_engine**() is used to discover the authoritative snmpEngineId of a remote SNMPv3 agent. A request PDU with empty USM user name is sent and the client's engine parameters are set according to the snmpEngine parameters received in the response PDU. If the client is configured to use authentication and/or privacy and the snmpEngineBoots and/or snmpEngineTime in the response had zero values, an additional request (possibly encrypted) with the appropriate user credentials is sent to fetch the missing values. Note, that the function blocks until the discovery process is completed. If no response could be received after all timeouts and retries, or the response contained errors the function returns -1. If the discovery process was completed 0 is returned.

The function **snmp_parse_server**() is used to parse an SNMP server specification string and fill in the fields of a *struct snmp_client*. The syntax of a server specification is

```
[trans::][community@][server][:port]
```

where *trans* is the transport name (one of "udp", "udp6", "stream" or "dgram"), *community* is the string to be used for both the read and the write community, *server* is the server's host name in case of UDP and the path name in case of a local socket, and *port* is the port in case of UDP transport. The function returns 0 in the case of success and return -1 and sets the error string in case of an error.

The function **snmp_parse_serverr**() fills the transport, the port number and the community strings with reasonable default values when they are not specified. The default transport is SNMP_TRANS_UDP. If the host name contains a slash the default is modified to SNMP_TRANS_LOC_DGRAM. If the host

name looks like a numeric IPv6 address the default is SNMP_TRANS_UDP6. For numeric IPv6 addresses the transport name udp is automatically translated as SNMP_TRANS_UDP6. The default port number (for udp or udp6) is "snmp". The default read community is "public" and the default write community "private".

snmp_parse_server() recognizes path names, host names and numerical IPv4 and IPv6 addresses. A string consisting of digits and periods is assumed to be an IPv4 address and must be parseable by **inet_aton**(*3*). An IPv6 address is any string enclosed in square brackets. It must be parseable with **gethostinfo**(*3*).

The port number for **snmp_parse_server**() can be specified numerically or symbolically. It is ignored for local sockets.

DIAGNOSTICS

If an error occurs in any of the functions an error indication as described above is returned. Additionally the function sets a printable error string in the *error* field of *snmp_client*.

SEE ALSO

gensnmptree(1), bsnmpd(1), bsnmpagent(3), bsnmplib(3)

STANDARDS

This implementation conforms to the applicable IETF RFCs and ITU-T recommendations.

AUTHORS

Hartmut Brandt <harti@FreeBSD.org>
Kendy Kutzner <kutzner@fokus.gmd.de>