## NAME

**bsnmpagent**, **snmp_depop_t**, **snmp_op_t**, **tree**, **tree_size**, **snmp_trace**, **snmp_debug**, **snmp_get**, **snmp_getnext**, **snmp_getbulk**, **snmp_set**, **snmp_make_errresp**, **snmp_dep_lookup**, **snmp_init_context**, **snmp_dep_commit**, **snmp_dep_rollback**, **snmp_dep_finish** - SNMP agent library

## LIBRARY

Begemot SNMP library (libbsnmp, -lbsnmp)

## SYNOPSIS

**#include <asn1.h>**
**#include <snmp.h>**
**#include <snmpagent.h>**

*typedef int*
**(*snmp_depop_t)**(*struct snmp_context *ctx, struct snmp_dependency *dep, enum snmp_depop op*);

*typedef int*
**(*snmp_op_t)**(*struct snmp_context *ctx, struct snmp_value *val, u_int len, u_int idx,*
*enum snmp_op op*);

*extern struct snmp_node *tree*;
*extern u_int tree_size*;
*extern u_int snmp_trace*;
*extern void (*snmp_debug)(const char *fmt, ...)*;

*enum snmp_ret*
**snmp_get**(*struct snmp_pdu *pdu, struct asn_buf *resp_b, struct snmp_pdu *resp, void *data*);

*enum snmp_ret*
**snmp_getnext**(*struct snmp_pdu *pdu, struct asn_buf *resp_b, struct snmp_pdu *resp, void *data*);

*enum snmp_ret*
**snmp_getbulk**(*struct snmp_pdu *pdu, struct asn_buf *resp_b, struct snmp_pdu *resp, void *data*);

*enum snmp_ret*
**snmp_set**(*struct snmp_pdu *pdu, struct asn_buf *resp_b, struct snmp_pdu *resp, void *data*);

*enum snmp_ret*
**snmp_make_errresp**(*const struct snmp_pdu *pdu, struct asn_buf *req_b, struct asn_buf *resp_b*);

*struct snmp_dependency **
**snmp_dep_lookup**(*struct snmp_context *ctx, const struct asn_oid *base, const struct asn_oid *idx, size_t alloc, snmp_depop_t func*);

*struct snmp_context **
**snmp_init_context**(*void*);

*int*
**snmp_dep_commit**(*struct snmp_context *ctx*);

*int*
**snmp_dep_rollback**(*struct snmp_context *ctx*);

*void*
**snmp_dep_finish**(*struct snmp_context *ctx*);

## DESCRIPTION

The SNMP library contains routines to easily build SNMP agent applications that use SNMP versions 1 or 2. Note, however, that it may be even easier to build an bsnmpd(1) loadable module, that handles the new MIB (see snmpmod(3)).

Most of the agent routines operate on a global array that the describes the complete MIB served by the agent. This array is held in the two variables:

```
extern struct snmp_node *tree;
extern u_int  tree_size;
```

The elements of the array are of type *struct snmp_node*:

```
typedef int (*snmp_op_t)(struct snmp_context *, struct snmp_value *,
    u_int, u_int, enum snmp_op);

struct snmp_node {
        struct asn_oid oid;
        const char          *name;          /* name of the leaf */
        enum snmp_node_type type;  /* type of this node */
        enum snmp_syntax syntax;
        snmp_op_t           op;
        u_int               flags;
        u_int32_t index;            /* index data */
```

```
        void            *data;          /* application data */
        void            *tree_data;     /* application data */
};
```

The fields of this structure are described below.

*oid* Base OID of the scalar or table column.

*name* Name of this variable.

*type* Type of this variable. One of:

```
enum snmp_node_type {
        SNMP_NODE_LEAF = 1,
        SNMP_NODE_COLUMN
};
```

*syntax* The SNMP syntax of this variable.

*op* The user supplied handler for this variable. The handler is called with the following arguments:

*ctx* A pointer to the context (see below). NULL.

*val* The value to be set or retrieved. For GETNEXT and GETBULK operations the oid in this value is the current OID. The function (called in this case only for table rows) must find the lexically next existing OID within the same column and set the oid and value subfields accordingly. If the table column is exhausted the function must return SNMP_ERR_NOSUCHNAME. For all other operations the oid in *val* is the oid to fetch or set.

*len* The length of the base oid without index.

*idx*
For table columns this is the index expression from the node (see below).

*op* This is the operation to execute, one of:

```
enum snmp_op {
        SNMP_OP_GET    = 1,
        SNMP_OP_GETNEXT,
```

```
                SNMP_OP_SET,
                SNMP_OP_COMMIT,
                SNMP_OP_ROLLBACK,
        };
```

The user handler must return an appropriate SNMP v2 error code. If the original PDU was a version 1 PDU, the error code is mapped automatically.

*flags* Currently only the flag SNMP_NODE_CANSET is defined and set for nodes, that can be written or created.

*index* This word describes the index for table columns. Each part of the index takes 4 bits starting at bit 4. Bits 0 to 3 hold the number of index parts. This arrangement allows for tables with up to seven indexes. Each bit group contains the syntax for the index part. There are a number of macros to help in parsing this field:

```
#define SNMP_INDEXES_MAX        7
#define SNMP_INDEX_SHIFT        4
#define SNMP_INDEX_MASK         0xf
#define SNMP_INDEX_COUNT(V)     ((V) & SNMP_INDEX_MASK)
#define SNMP_INDEX(V,I) \
        (((V) >> (((I) + 1) * SNMP_INDEX_SHIFT)) & \
        SNMP_INDEX_MASK)
```

*data* This field may contain arbitrary data and is not used by the library.

The easiest way to construct the node table is gensnmptree(1). Note, that one must be careful when changing the tree while executing a SET operation. Consult the sources for bsnmpd(1).

The global variable *snmp_trace* together with the function pointed to by *snmp_debug* help in debugging the library and the agent. *snmp_trace is a bit mask with the following bits:*

```
enum {
        SNMP_TRACE_GET,
        SNMP_TRACE_GETNEXT,
        SNMP_TRACE_SET,
        SNMP_TRACE_DEPEND,
        SNMP_TRACE_FIND,
};
```

Setting a bit to true causes the library to call **snmp_debug**() in strategic places with a debug string. The library contains a default implementation for the debug function that prints a message to standard error.

Many of the functions use a so called context:

```
struct snmp_context {
        u_int       var_index;
        struct snmp_scratch *scratch;
        struct snmp_dependency *dep;
        void        *data;              /* user data */
        enum snmp_ret code;             /* return code */
};

struct snmp_scratch {
        void                *ptr1;
        void                *ptr2;
        uint32_t    int1;
        uint32_t    int2;
};
```

The fields are used as follows:

*va_index* For the node operation callback this is the index of the variable binding that should be returned if an error occurs. Set by the library. In all other functions this is undefined.

*scratch* For the node operation callback this is a pointer to a per variable binding scratch area that can be used to implement the commit and rollback. Set by the library.

*dep* In the dependency callback function (see below) this is a pointer to the current dependency. Set by the library.

*data* This is the *data* argument from the call to the library and is not used by the library.

The next three functions execute different kinds of GET requests. The function **snmp_get**() executes an SNMP GET operation, the function **snmp_getnext**() executes an SNMP GETNEXT operation and the function **snmp_getbulk**() executes an SNMP GETBULK operation. For all three functions the response PDU is constructed and encoded on the fly. If everything is ok, the response PDU is returned in *resp* and *resp_b*. The caller must call **snmp_pdu_free**() to free the response PDU in this case. One of the following values may be returned:

SNMP_RET_OK Operation successful, response PDU may be sent.

SNMP_RET_IGN Operation failed, no response PDU constructed. Request is ignored.

SNMP_RET_ERR Error in operation. The error code and index have been set in *pdu*. No response PDU has been constructed. The caller may construct an error response PDU via **snmp_make_errresp**().

The function **snmp_set**() executes an SNMP SET operation. The arguments are the same as for the previous three functions. The operation of this functions is, however, much more complex.

The SET operation occurs in several stages:

1. For each binding search the corresponding nodes, check that the variable is writeable and the syntax is ok. The writeable check can be done only for scalars. For columns it must be done in the node's operation callback function.

2. For each binding call the node's operation callback with function SNMP_OP_SET. The callback may create dependencies or finalizers (see below). For simple scalars the scratch area may be enough to handle commit and rollback, for interdependent table columns dependencies may be necessary.

3. If the previous step fails at any point, the node's operation callback functions are called for all bindings for which SNMP_OP_SET was executed with SNMP_OP_ROLLBACK, in the opposite order. This allows all variables to undo the effect of the SET operation. After this all the dependencies are freed and the finalizers are executed with a fail flag of 1. Then the function returns to the caller with an appropriate error indication.

4. If the SET step was successful for all bindings, the dependency callbacks are executed in the order in which the dependencies were created with an operation of SNMP_DEPOP_COMMIT. If any of the dependencies fails, all the committed dependencies are called again in the opposite order with SNMP_DEPOP_ROLLBACK. Than for all bindings from the last to the first the node's operation callback is called with SNMP_OP_ROLLBACK to undo the effect of SNMP_OP_SET. At the end the dependencies are freed and the finalizers are called with a fail flag of 1 and the function returns to the caller with an appropriate error indication.

5. If the dependency commits were successful, for each binding the node's operation callback is called with SNMP_OP_COMMIT. Any error returned from the callbacks is ignored (an error message is generated via **snmp_error**().)

6. Now the dependencies are freed and the finalizers are called with a fail flag of 0. For each dependency just before freeing it its callback is called with SNMP_DEPOP_FINISH. Then the function returns SNMP_ERR_OK.

There are to mechanisms to help in complex SET operations: dependencies and finalizers. A dependency is used if several bindings depend on each other. A typical example is the creation of a conceptual row, which requires the setting of several columns to succeed. A dependency is identified by two OIDs. In the table case, the first oid is typically the table's base OID and the second one the index. Both of these can easily be generated from the variables OID with **asn_slice_oid**(). The function **snmp_dep_lookup**() tries to find a dependency based on these two OIDs and, if it cannot find one creates a new one. This means for the table example, that the function returns the same dependency for each of the columns of the same table row. This allows during the SNMP_OP_SET processing to collect all information about the row into the dependency. The arguments to **snmp_dep_lookup**() are: the two OIDs to identify the dependency (they are copied into newly created dependencies), the size of the structure to allocate and the dependency callback.

When all SNMP_OP_SET operations have succeeded the dependencies are executed. At this stage the dependency callback has all information about the given table row that was available in this SET PDU and can operate accordingly.

It is guaranteed that each dependency callback is executed at minimum once - with an operation of SNMP_OP_ROLLBACK. This ensures that all dynamically allocated resources in a callback can be freed correctly.

The function **snmp_make_errresp**() makes an error response if an operation has failed. It takes the original request PDU (it will look only on the error code and index fields), the buffer containing the original PDU and a buffer for the error PDU. It copies the bindings field from the original PDUs buffer directly to the response PDU and thus does not depend on the decodability of this field. It may return the same values as the operation functions.

The next four functions allow some parts of the SET operation to be executed. This is only used in bsnmpd(1) to implement the configuration as a single transaction. The function **snmp_init_context**() creates and initializes a context. The function **snmp_dep_commit**() executes SNMP_DEPOP_COMMIT for all dependencies in the context stopping at the first error. The function **snmp_dep_rollback**() executes SNMP_DEPOP_ROLLBACK starting at the previous of the current dependency in the context. The function **snmp_dep_finish**() executes SNMP_DEPOP_FINISH for all dependencies.

## DIAGNOSTICS

If an error occurs in any of the function an error indication as described above is returned. Additionally the functions may call snmp_error on unexpected errors.

## SEE ALSO

gensnmptree(1), bsnmpd(1), bsnmpclient(3), bsnmplib(3), snmpmod(3)

## STANDARDS

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

## AUTHORS

Hartmut Brandt <harti@FreeBSD.org>