NAME

INSERT_OBJECT_OID_LINK_INDEX, **INSERT_OBJECT_INT_LINK_INDEX**, **FIND_OBJECT_OID_LINK_INDEX**, **NEXT_OBJECT_OID_LINK_INDEX**, **FIND_OBJECT_INT_LINK_INDEX**, **NEXT_OBJECT_INT_LINK_INDEX**, **INSERT_OBJECT_OID_LINK**, **INSERT_OBJECT_INT_LINK**, **FIND_OBJECT_OID_LINK**, **NEXT_OBJECT_OID_LINK**, **FIND_OBJECT_INT_LINK**, **NEXT_OBJECT_INT_LINK**, **INSERT_OBJECT_OID**, **INSERT_OBJECT_INT**, **FIND_OBJECT_OID**, **FIND_OBJECT_INT**, **NEXT_OBJECT_OID**, **NEXT_OBJECT_INT**, **this_tick**, **start_tick**, **get_ticks**, **systemg**, **comm_define**, **community**, **oid_zeroDotZero**, **oid_usmUnknownEngineIDs**, **oid_usmNotInTimeWindows**, **reqid_allocate**, **reqid_next**, **reqid_base**, **reqid_istype**, **reqid_type**, **timer_start**, **timer_start_repeat**, **timer_stop**, **fd_select**, **fd_deselect**, **fd_suspend**, **fd_resume**, **or_register**, **or_unregister**, **buf_alloc**, **buf_size**, **snmp_input_start**, **snmp_input_finish**, **snmp_output**, **snmp_send_port**, **snmp_send_trap**, **snmp_pdu_auth_access string_save**, **string_commit**, **string_rollback**, **string_get**, **string_get_max**, **string_free**, **ip_save**, **ip_rollback**, **ip_commit**, **ip_get**, **oid_save**, **oid_rollback**, **oid_commit**, **oid_get**, **index_decode**, **index_compare**, **index_compare_off**, **index_append**, **index_append_off**, **snmpd_usmstats**, **bsnmpd_get_usm_stats**, **bsnmpd_reset_usm_stats**, **usm_first_user**, **usm_next_user**, **usm_find_user**, **usm_new_user**, **usm_delete_user**, **usm_flush_users**, **usm_user**, **snmpd_target_stat**, **bsnmpd_get_target_stats**, **target_first_address**, **target_next_address**, **target_new_address**, **target_activate_address**, **target_delete_address**, **target_first_param**, **target_next_param**, **target_new_param**, **target_delete_param**, **target_first_notify**, **target_next_notify**, **target_new_notify**, **target_delete_notify**, **target_flush_all**, **target_address**, **target_param**, **target_notify** - SNMP daemon loadable module interface

LIBRARY

Begemot SNMP library (libbsnmp, -lbsnmp)

SYNOPSIS

#include <bsnmp/snmpmod.h>

INSERT_OBJECT_OID_LINK_INDEX(*PTR*, *LIST*, *LINK*, *INDEX*);

INSERT_OBJECT_INT_LINK_INDEX(*PTR*, *LIST*, *LINK*, *INDEX*);

FIND_OBJECT_OID_LINK_INDEX(*LIST*, *OID*, *SUB*, *LINK*, *INDEX*);

FIND_OBJECT_INT_LINK_INDEX(*LIST*, *OID*, *SUB*, *LINK*, *INDEX*);

NEXT_OBJECT_OID_LINK_INDEX(*LIST*, *OID*, *SUB*, *LINK*, *INDEX*);

NEXT_OBJECT_INT_LINK_INDEX(*LIST*, *OID*, *SUB*, *LINK*, *INDEX*);

INSERT_OBJECT_OID_LINK(*PTR, LIST, LINK*);

INSERT_OBJECT_INT_LINK(*PTR, LIST, LINK*);

FIND_OBJECT_OID_LINK(*LIST, OID, SUB, LINK*);

FIND_OBJECT_INT_LINK(*LIST, OID, SUB, LINK*);

NEXT_OBJECT_OID_LINK(*LIST, OID, SUB, LINK*);

NEXT_OBJECT_INT_LINK(*LIST, OID, SUB, LINK*);

INSERT_OBJECT_OID(*PTR, LIST*);

INSERT_OBJECT_INT(*PTR, LIST*);

FIND_OBJECT_OID(*LIST, OID, SUB*);

FIND_OBJECT_INT(*LIST, OID, SUB*);

NEXT_OBJECT_OID(*LIST, OID, SUB*);

NEXT_OBJECT_INT(*LIST, OID, SUB*);

extern uint64_t this_tick;
extern uint64_t start_tick;

uint64_t
get_ticks(*void*);

extern struct systemg systemg;

u_int
comm_define(*u_int priv, const char *descr, struct lmodule *mod, const char *str*);

*const char **
comm_string(*u_int comm*);

extern u_int community;
extern const struct asn_oid oid_zeroDotZero;

u_int
reqid_allocate(*int size, struct lmodule *mod*);

int32_t
reqid_next(*u_int type*);

int32_t
reqid_base(*u_int type*);

int
reqid_istype(*int32_t reqid, u_int type*);

u_int
reqid_type(*int32_t reqid*);

*void **
timer_start(*u_int ticks, void (*func)(void *), void *uarg, struct lmodule *mod*);

*void **
timer_start_repeat(*u_int ticks, u_int repeat_ticks, void (*func)(void *), void *uarg,*
*struct lmodule *mod*);

void
timer_stop(*void *timer_id*);

*void **
fd_select(*int fd, void (*func)(int, void *), void *uarg, struct lmodule *mod*);

void
fd_deselect(*void *fd_id*);

void
fd_suspend(*void *fd_id*);

int
fd_resume(*void *fd_id*);

u_int
or_register(*const struct asn_oid *oid, const char *descr, struct lmodule *mod*);

void
or_unregister(*u_int or_id*);

*void **
buf_alloc(*int tx*);

size_t
buf_size(*int tx*);

enum snmpd_input_err
snmp_input_start(*const u_char *buf, size_t len, const char *source, struct snmp_pdu *pdu, int32_t *ip, size_t *pdulen*);

enum snmpd_input_err
snmp_input_finish(*struct snmp_pdu *pdu, const u_char *rcvbuf, size_t rcvlen, u_char *sndbuf, size_t *sndlen, const char *source, enum snmpd_input_err ierr, int32_t ip, void *data*);

void
snmp_output(*struct snmp_pdu *pdu, u_char *sndbuf, size_t *sndlen, const char *dest*);

void
snmp_send_port(*void *trans, const struct asn_oid *port, struct snmp_pdu *pdu, const struct sockaddr *addr, socklen_t addrlen*);

void
snmp_send_trap(*const struct asn_oid *oid, ...*);

enum snmp_code
snmp_pdu_auth_access(*struct snmp_pdu *pdu, int32_t *ip*);

int
string_save(*struct snmp_value *val, struct snmp_context *ctx, ssize_t req_size, u_char **strp*);

void
string_commit(*struct snmp_context *ctx*);

void
string_rollback(*struct snmp_context *ctx, u_char **strp*);

int

string_get(*struct snmp_value *val, const u_char *str, ssize_t len*);

int
string_get_max(*struct snmp_value *val, const u_char *str, ssize_t len, size_t maxlen*);

void
string_free(*struct snmp_context *ctx*);

int
ip_save(*struct snmp_value *val, struct snmp_context *ctx, u_char *ipa*);

void
ip_rollback(*struct snmp_context *ctx, u_char *ipa*);

void
ip_commit(*struct snmp_context *ctx*);

int
ip_get(*struct snmp_value *val, u_char *ipa*);

int
oid_save(*struct snmp_value *val, struct snmp_context *ctx, struct asn_oid *oid*);

void
oid_rollback(*struct snmp_context *ctx, struct asn_oid *oid*);

void
oid_commit(*struct snmp_context *ctx*);

int
oid_get(*struct snmp_value *val, const struct asn_oid *oid*);

int
index_decode(*const struct asn_oid *oid, u_int sub, u_int code, ...*);

int
index_compare(*const struct asn_oid *oid1, u_int sub, const struct asn_oid *oid2*);

int
index_compare_off(*const struct asn_oid *oid1, u_int sub, const struct asn_oid *oid2, u_int off*);

void
index_append(*struct asn_oid *dst, u_int sub, const struct asn_oid *src*);

void
index_append_off(*struct asn_oid *dst, u_int sub, const struct asn_oid *src, u_int off*);

extern struct snmpd_usmstat snmpd_usmstats;

*struct snmpd_usmstat **
bsnmpd_get_usm_stats(*void*);

void
bsnmpd_reset_usm_stats(*void*);

*struct usm_user **
usm_first_user(*void*);

*struct usm_user **
usm_next_user(*struct usm_user *uuser*);

*struct usm_user **
usm_find_user(*uint8_t *engine, uint32_t elen, char *uname*);

*struct usm_user **
usm_new_user(*uint8_t *engine, uint32_t elen, char *uname*);

void
usm_delete_user(*struct usm_user **);

void
usm_flush_users(*void*);

*extern struct usm_user *usm_user;*

*struct snmpd_target_stats **
bsnmpd_get_target_stats(*void*);

*struct target_address **
target_first_address(*void*);

*struct target_address **
target_next_address(*struct target_address **);

*struct target_address **
target_new_address(*char **);

int
target_activate_address(*struct target_address **);

int
target_delete_address(*struct target_address **);

*struct target_param **
target_first_param(*void*);

*struct target_param **
target_next_param(*struct target_param **);

*struct target_param **
target_new_param(*char **);

int
target_delete_param(*struct target_param **);

*struct target_notify **
target_first_notify(*void*);

*struct target_notify **
target_next_notify(*struct target_notify **);

*struct target_notify **
target_new_notify(*char **);

int
target_delete_notify(*struct target_notify **);

void
target_flush_all(*void*);

extern const struct asn_oid oid_usmUnknownEngineIDs;

extern const struct asn_oid oid_usmNotInTimeWindows;

DESCRIPTION

The bsnmpd(1) SNMP daemon implements a minimal MIB which consists of the system group, part of the SNMP MIB, a private configuration MIB, a trap destination table, a UDP port table, a community table, a module table, a statistics group and a debugging group. All other MIBs are support through loadable modules. This allows bsnmpd(1) to use for task, that are not the classical SNMP task.

MODULE LOADING AND UNLOADING

Modules are loaded by writing to the module table. This table is indexed by a string, that identifies the module to the daemon. This identifier is used to select the correct configuration section from the configuration files and to identify resources allocated to this module. A row in the module table is created by writing a string of non-zero length to the *begemotSnmpdModulePath* column. This string must be the complete path to the file containing the module. A module can be unloaded by writing a zero length string to the path column of an existing row.

Modules may depend on each other an hence must be loaded in the correct order. The dependencies are listed in the corresponding manual pages.

Upon loading a module the SNMP daemon expects the module file to a export a global symbol *config*. This symbol should be a variable of type *struct snmp_module*:

```
typedef enum snmpd_proxy_err (*proxy_err_f)(struct snmp_pdu *, void *,
    const struct asn_oid *, const struct sockaddr *, socklen_t,
    enum snmpd_input_err, int32_t);


struct snmp_module {
        const char *comment;
        int (*init)(struct lmodule *, int argc, char *argv[]);
        int (*fini)(void);
        void (*idle)(void);
        void (*dump)(void);
        void (*config)(void);
        void (*start)(void);
        proxy_err_f proxy;
        const struct snmp_node *tree;
        u_int tree_size;
        void (*loading)(const struct lmodule *, int);
};
```

This structure must be statically initialized and its fields have the following functions:

comment This is a string that will be visible in the module table. It should give some hint about the function of this module.

init This function is called upon loading the module. The module pointer should be stored by the module because it is needed in other calls and the argument vector will contain the arguments to this module from the daemons command line. This function should return 0 if everything is ok or an UNIX error code (see errno(3)). Once the function returns 0, the *fini* function is called when the module is unloaded.

fini The module is unloaded. This gives the module a chance to free resources that are not automatically freed. Be sure to free all memory, because daemons tend to run very long. This function pointer may be NULL if it is not needed.

idle If this function pointer is not NULL, the function pointed to by it is called whenever the daemon is going to wait for an event. Try to avoid using this feature.

dump Whenever the daemon receives a SIGUSR1 it dumps it internal state via syslog(3). If the *dump* field is not NULL it is called by the daemon to dump the state of the module.

config Whenever the daemon receives a SIGHUP signal it re-reads its configuration file. If the *config* field is not NULL it is called after reading the configuration file to give the module a chance to adapt to the new configuration.

start If not NULL this function is called after successful loading and initializing the module to start its actual operation.

proxy If the daemon receives a PDU and that PDU has a community string whose community was registered by this module and *proxy* is not NULL than this function is called to handle the PDU.

tree This is a pointer to the node array for the MIB tree implemented by this module.

tree_size This is the number of nodes in *tree*.

loading If this pointer is not NULL it is called whenever another module was loaded or unloaded. It gets a pointer to that module and a flag that is 0 for unloading and 1 for loading.

When everything is ok, the daemon merges the module’s MIB tree into its current global tree, calls the

modules **init**() function. If this function returns an error, the modules MIB tree is removed from the global one and the module is unloaded. If initialization is successful, the modules **start**() function is called. After it returns the **loaded**() functions of all modules (including the loaded one) are called.

When the module is unloaded, its MIB tree is removed from the global one, the communities, request id ranges, running timers and selected file descriptors are released, the **fini**() function is called, the module file is unloaded and the **loaded**() functions of all other modules are called.

IMPLEMENTING TABLES

There are a number of macros designed to help implementing SNMP tables. A problem while implementing a table is the support for the GETNEXT operator. The GETNEXT operation has to find out whether, given an arbitrary OID, the lessest table row, that has an OID higher than the given OID. The easiest way to do this is to keep the table as an ordered list of structures each one of which contains an OID that is the index of the table row. This allows easy removal, insertion and search.

The helper macros assume, that the table is organized as a TAILQ (see queue(3) and each structure contains a *struct asn_oid* that is used as index. For simple tables with only a integer or unsigned index, an alternate form of the macros is available, that presume the existence of an integer or unsigned field as index field.

The macros have name of the form

{INSERT,FIND,NEXT}_OBJECT_{OID,INT}[_LINK[_INDEX]]

The **INSERT_***() macros are used in the SET operation to insert a new table row into the table. The **FIND_***() macros are used in the GET operation to find a specific row in the table. The **NEXT_***() macros are used in the GETNEXT operation to find the next row in the table. The last two macros return a pointer to the row structure if a row is found, NULL otherwise. The macros ***_OBJECT_OID_***() assume the existence of a *struct asn_oid* that is used as index, the macros ***_OBJECT_INT_***() assume the existence of an unsigned integer field that is used as index.

The macros ***_INDEX**() allow the explicit naming of the index field in the parameter *INDEX*, whereas the other macros assume that this field is named *index*. The macros ***_LINK_***() allow the explicit naming of the link field of the tail queues, the others assume that the link field is named *link*. Explicitly naming the link field may be necessary if the same structures are held in two or more different tables.

The arguments to the macros are as follows:

PTR A pointer to the new structure to be inserted into the table.

LIST A pointer to the tail queue head.

LINK The name of the link field in the row structure.

INDEX The name of the index field in the row structure.

OID Must point to the *var* field of the *value* argument to the node operation callback. This is the OID to search for.

SUB This is the index of the start of the table index in the OID pointed to by *OID*. This is usually the same as the *sub* argument to the node operation callback.

DAEMON TIMESTAMPS

The variable *this_tick* contains the tick (there are 100 SNMP ticks in a second) when the current PDU processing was started. The variable *start_tick* contains the tick when the daemon was started. The function **get_ticks**() returns the current tick. The number of ticks since the daemon was started is

```
get_ticks() - start_tick
```

THE SYSTEM GROUP

The scalar fields of the system group are held in the global variable *systemg*:

```
struct systemg {
        u_char          *descr;
        struct asn_oid  object_id;
        u_char          *contact;
        u_char          *name;
        u_char          *location;
        uint32_t  services;
        uint32_t  or_last_change;
};
```

COMMUNITIES

The SNMP daemon implements a community table. On receipt of a request message the community string in that message is compared to each of the community strings in that table, if a match is found, the global variable *community* is set to the community identifier for that community. Community identifiers are unsigned integers. For the three standard communities there are three constants defined:

```
#define COMM_INITIALIZE         0
#define COMM_READ       1
```

```
#define COMM_WRITE      2
```

community is set to COMM_INITIALIZE while the assignments in the configuration file are processed. To COMM_READ or COMM_WRITE when the community strings for the read-write or read-only community are found in the incoming PDU.

Modules can define additional communities. This may be necessary to provide transport proxying (a PDU received on one communication link is proxied to another link) or to implement non-UDP access points to SNMP. A new community is defined with the function **comm_define**(). It takes the following parameters:

priv This is an integer identifying the community to the module. Each module has its own namespace with regard to this parameter. The community table is indexed with the module name and this identifier.

descr This is a string providing a human readable description of the community. It is visible in the community table.

mod This is the module defining the community.

str This is the initial community string.

The function returns a globally unique community identifier. If a SNMPv1 or SNMPv2 PDU is received who's community string matches, this identifier is set into the global *community*.

The function **comm_string**() returns the current community string for the given community.

All communities defined by a module are automatically released when the module is unloaded.

THE USER-BASED SECURITY GROUP

The scalar statistics of the USM group are held in the global variable *snmpd_usmstats*:

```
struct snmpd_usmstat {
        uint32_t  unsupported_seclevels;
        uint32_t  not_in_time_windows;
        uint32_t  unknown_users;
        uint32_t  unknown_engine_ids;
        uint32_t  wrong_digests;
        uint32_t  decrypt_errors;
};
```

bsnmpd_get_usm_stats() returns a pointer to the global structure containing the statistics. **bsnmpd_reset_usm_stats**() clears the statistics of the USM group.

A global list of configured USM users is maintained by the daemon.

```
struct usm_user {
        struct snmp_user        suser;
        uint8_t                 user_engine_id[SNMP_ENGINE_ID_SIZ];
        uint32_t                user_engine_len;
        char                    user_public[SNMP_ADM_STR32_SIZ];
        uint32_t                user_public_len;
        int32_t                 status;
        int32_t                 type;
        SLIST_ENTRY(usm_user)   up;
};
```

This structure represents an USM user. The daemon only responds to SNMPv3 PDUs with user credentials matching an USM user entry in its global list. If a SNMPv3 PDU is received, whose security model is USM, the global *usm_user* is set to point at the user entry that matches the credentials contained in the PDU. However, the daemon does not create or remove USM users, it gives an interface to external loadable module(s) to manage the list. **usm_new_user**() adds an user entry in the list, and **usm_delete_user**() deletes an existing entry from the list. **usm_flush_users**() is used to remove all configured USM users. **usm_first_user**() will return the first user in the list, or NULL if the list is empty. **usm_next_user**() will return the next user of a given entry if one exists, or NULL. The list is sorted according to the USM user name and Engine ID. **usm_find_user**() returns the USM user entry matching the given *engine* and *uname* or NULL if an user with the specified name and engine id is not present in the list.

THE MANAGEMENT TARGET GROUP

The Management Target group holds target address information used when sending SNMPv3 notifications.

The scalar statistics of the Management Target group are held in the global variable *snmpd_target_stats*:

```
struct snmpd_target_stats {
        uint32_t        unavail_contexts;
        uint32_t        unknown_contexts;
};
```

bsnmpd_get_target_stats() returns a pointer to the global structure containing the statistics.

Three global lists of configured management target addresses, parameters and notifications respectively

are maintained by the daemon.

```
struct target_address {
        char                            name[SNMP_ADM_STR32_SIZ];
        uint8_t                         address[SNMP_UDP_ADDR_SIZ];
        int32_t                         timeout;
        int32_t                         retry;
        char                            taglist[SNMP_TAG_SIZ];
        char                            paramname[SNMP_ADM_STR32_SIZ];
        int32_t                         type;
        int32_t                         socket;
        int32_t                         status;
        SLIST_ENTRY(target_address)     ta;
};
```

This structure represents a SNMPv3 Management Target address. Each time a SNMP TRAP is send the daemon will send the Trap to all active Management Target addresses in its global list.

```
struct target_param {
        char                            name[SNMP_ADM_STR32_SIZ];
        int32_t                         mpmodel;
        int32_t                         sec_model;
        char                            secname[SNMP_ADM_STR32_SIZ];
        enum snmp_usm_level             sec_level;
        int32_t                         type;
        int32_t                         status;
        SLIST_ENTRY(target_param)       tp;
};
```

This structure represents the information used to generate SNMP messages to the associated SNMPv3 Management Target addresses.

```
struct target_notify {
        char                            name[SNMP_ADM_STR32_SIZ];
        char                            taglist[SNMP_TAG_SIZ];
        int32_t                         notify_type;
        int32_t                         type;
        int32_t                         status;
        SLIST_ENTRY(target_notify)      tn;
};
```

This structure represents Notification Tag entries - SNMP notifications are sent to the Target address for each entry in the Management Target Address list that has a tag matching the specified tag in this

structure.

The daemon does not create or remove entries in the Management Target group lists, it gives an interface to external loadable module(s) to manage the lists. **target_new_address**() adds a target address entry, and **target_delete_address**() deletes an existing entry from the target address list. **target_activate_address**() creates a socket associated with the target address entry so that SNMP notifications may actually be send to that target address. **target_first_address**() will return a pointer to the first target address entry in the list, while **target_next_address**() will return a pointer to the next target address of a given entry if one exists. **target_new_param**() adds a target parameters' entry, and **target_delete_param**() deletes an existing entry from the target parameters list. **target_first_param**() will return a pointer to the first target parameters' entry in the list, while **target_next_param**() will return a pointer to the next target parameters of a given entry if one exists. **target_new_notify**() adds a notification target entry, and **target_delete_notify**() deletes an existing entry from the notification target list. **target_first_notify**() will return a pointer to the first notification target entry in the list, while **target_next_notify**() will return a pointer to the next notification target of a given entry if one exists. **target_flush_all**() is used to remove all configured data from the three global Management Target Group lists.

WELL KNOWN OIDS

The global variable *oid_zeroDotZero* contains the OID 0.0. The global variables *oid_usmUnknownEngineIDs oid_usmNotInTimeWindows* contains the OIDs 1.3.6.1.6.3.15.1.1.4.0 and 1.3.6.1.6.3.15.1.1.2.0 used in the SNMPv3 USM Engine Discovery.

REQUEST ID RANGES

For modules that implement SNMP client functions besides SNMP agent functions it may be necessary to identify SNMP requests by their identifier to allow easier routing of responses to the correct sub-system. Request id ranges provide a way to acquire globally non-overlapping sub-ranges of the entire 31-bit id range.

A request id range is allocated with **reqid_allocate**(). The arguments are: the size of the range and the module allocating the range. For example, the call

```
id = reqid_allocate(1000, module);
```

allocates a range of 1000 request ids. The function returns the request id range identifier or 0 if there is not enough identifier space. The function **reqid_base**() returns the lowest request id in the given range.

Request id are allocated starting at the lowest one linear throughout the range. If the client application may have a lot of outstanding request the range must be large enough so that an id is not reused until it is really expired. **reqid_next**() returns the sequentially next id in the range.

The function **reqid_istype**() checks whether the request id *reqid* is within the range identified by *type*. The function **reqid_type**() returns the range identifier for the given *reqid* or 0 if the request id is in none of the ranges.

TIMERS

The SNMP daemon supports an arbitrary number of timers with SNMP tick granularity. The function **timer_start**() arranges for the callback *func* to be called with the argument *uarg* after *ticks* SNMP ticks have expired. *mod* is the module that starts the timer. These timers are one-shot, they are not restarted. Repeatable timers are started with **timer_start_repeat**() which takes an additional argument *repeat_ticks*. The argument *ticks* gives the number of ticks until the first execution of the callback, while *repeat_ticks* is the number of ticks between invocations of the callback. Note, that currently the number of initial ticks silently may be set identical to the number of ticks between callback invocations. The function returns a timer identifier that can be used to stop the timer via **timer_stop**(). If a module is unloaded all timers started by the module that have not expired yet are stopped.

FILE DESCRIPTOR SUPPORT

A module may need to get input from socket file descriptors without blocking the daemon (for example to implement alternative SNMP transports).

The function **fd_select**() causes the callback function *func* to be called with the file descriptor *fd* and the user argument *uarg* whenever the file descriptor *fd* can be read or has a close condition. If the file descriptor is not in non-blocking mode, it is set to non-blocking mode. If the callback is not needed anymore, **fd_deselect**() may be called with the value returned from **fd_select**(). All file descriptors selected by a module are automatically deselected when the module is unloaded.

To temporarily suspend the file descriptor registration **fd_suspend**() can be called. This also causes the file descriptor to be switched back to blocking mode if it was blocking prior the call to **fd_select**(). This is necessary to do synchronous input on a selected socket. The effect of **fd_suspend**() can be undone with **fd_resume**().

OBJECT RESOURCES

The system group contains an object resource table. A module may create an entry in this table by calling **or_register**() with the *oid* to be registered, a textual description in *str* and a pointer to the module *mod*. The registration can be removed with **or_unregister**(). All registrations of a module are automatically removed if the module is unloaded.

TRANSMIT AND RECEIVE BUFFERS

A buffer is allocated via **buf_alloc**(). The argument must be 1 for transmit and 0 for receive buffers. The function may return NULL if there is no memory available. The current buffersize can be obtained with **buf_size**().

PROCESSING PDUS

For modules that need to do their own PDU processing (for example for proxying) the following functions are available:

Function **snmp_input_start**() decodes the PDU, searches the community, and sets the global *this_tick*. It returns one of the following error codes:

SNMPD_INPUT_OK	Everything ok, continue with processing.
SNMPD_INPUT_FAILED	The PDU could not be decoded, has a wrong version or an unknown community string.
SNMPD_INPUT_VALBADLEN	A SET PDU had a value field in a binding with a wrong length field in an ASN.1 header.
SNMPD_INPUT_VALRANGE	A SET PDU had a value field in a binding with a value that is out of range for the given ASN.1 type.
SNMPD_INPUT_VALBADENC	A SET PDU had a value field in a binding with wrong ASN.1 encoding.
SNMPD_INPUT_TRUNC	The buffer appears to contain a valid begin of a PDU, but is too short. For streaming transports this means that the caller must save what he already has and trying to obtain more input and reissue this input to the function. For datagram transports this means that part of the datagram was lost and the input should be ignored.

The function **snmp_input_finish**() does the other half of processing: if **snmp_input_start**() did not return OK, tries to construct an error response. If the start was OK, it calls the correct function from bsnmpagent(3) to execute the request and depending on the outcome constructs a response or error response PDU or ignores the request PDU. It returns either SNMPD_INPUT_OK or SNMPD_INPUT_FAILED. In the first case a response PDU was constructed and should be sent.

The function **snmp_output**() takes a PDU and encodes it.

The function **snmp_send_port**() takes a PDU, encodes it and sends it through the given port (identified by the transport and the index in the port table) to the given address.

The function **snmp_send_trap**() sends a trap to all trap destinations. The arguments are the *oid*

identifying the trap and a NULL-terminated list of *struct snmp_value* pointers that are to be inserted into the trap binding list. **snmp_pdu_auth_access**() verifies whether access to the object IDs contained in the *pdu*
should be granted or denied, according to the configured View-Based Access rules. *ip* contains the index of the first varbinding to which access was denied, or 0 if access to all varbindings in the PDU is granted.

SIMPLE ACTION SUPPORT

For simple scalar variables that need no dependencies a number of support functions is available to handle the set, commit, rollback and get.

The following functions are used for OCTET STRING scalars, either NUL terminated or not:

string_save() should be called for SNMP_OP_SET. *value* and *ctx* are the resp. arguments to the node callback. *valp* is a pointer to the pointer that holds the current value and *req_size* should be -1 if any size of the string is acceptable or a number larger or equal zero if the string must have a specific size. The function saves the old value in the scratch area (note, that any initial value must have been allocated by malloc(3)), allocates a new string, copies over the new value, NUL-terminates it and sets the new current value.

string_commit()
simply frees the saved old value in the scratch area.

string_rollback()
frees the new value, and puts back the old one.

string_get() is used for GET or GETNEXT. The function

string_get_max()
can be used instead of **string_get**() to ensure that the returned string has a certain maximum length. If *len* is -1, the length is computed via strlen(3) from the current string value. If the current value is NULL, a OCTET STRING of zero length is returned.

string_free() must be called if either rollback or commit fails to free the saved old value.

The following functions are used to process scalars of type IP-address:

ip_save() Saves the current value in the scratch area and sets the new value from *valp*.

ip_commit() Does nothing.

ip_rollback() Restores the old IP address from the scratch area.

ip_get() Retrieves the IP current address.

The following functions handle OID-typed variables:

oid_save() Saves the current value in the scratch area by allocating a *struct asn_oid* with malloc(3) and sets the new value from *oid*.

oid_commit() Frees the old value in the scratch area.

oid_rollback() Restores the old OID from the scratch area and frees the old OID.

oid_get() Retrieves the OID

TABLE INDEX HANDLING

The following functions help in handling table indexes:

index_decode() Decodes the index part of the OID. The parameter *oid* must be a pointer to the *var* field of the *value* argument of the node callback. The *sub* argument must be the index of the start of the index in the OID (this is the *sub* argument to the node callback). *code* is the index expression (parameter *idx* to the node callback). These parameters are followed by parameters depending on the syntax of the index elements as follows:

INTEGER *int32_t* * expected as argument.

COUNTER64 *uint64_t* * expected as argument. Note, that this syntax is illegal for indexes.

OCTET STRING A *u_char* ** and a *size_t* * expected as arguments. A buffer is allocated to hold the decoded string.

OID A *struct asn_oid* * is expected as argument.

IP ADDRESS A *u_int8_t* * expected as argument that points to a buffer of at least four byte.

COUNTER, GAUGE, TIMETICKS

A *u_int32_t* expected.

NULL No argument expected.

index_compare()
compares the current variable with an OID. *oid1* and *sub* come from the node callback arguments *value->var* and *sub* resp. *oid2* is the OID to compare to. The function returns -1, 0, +1 when the variable is lesser, equal, higher to the given OID. *oid2* must contain only the index part of the table column.

index_compare_off()
is equivalent to **index_compare**() except that it takes an additional parameter *off* that causes it to ignore the first *off* components of both indexes.

index_append() appends OID *src* beginning at position *sub* to *dst*.

index_append_off()
appends OID *src* beginning at position *off* to *dst* beginning at position *sub* + *off*.

SEE ALSO

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

STANDARDS

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

AUTHORS

Hartmut Brandt <harti@FreeBSD.org>