NAME

nvlist_create, **nvlist_destroy**, **nvlist_error**, **nvlist_set_error**, **nvlist_empty**, **nvlist_flags**, **nvlist_exists**, **nvlist_free**, **nvlist_clone**, **nvlist_dump**, **nvlist_fdump**, **nvlist_size**, **nvlist_pack**, **nvlist_unpack**, **nvlist_send**, **nvlist_recv**, **nvlist_xfer**, **nvlist_in_array**, **nvlist_next**, **nvlist_add**, **nvlist_move**, **nvlist_get**, **nvlist_take**, **nvlist_append** - library for name/value pairs

LIBRARY

Name/value pairs library (libnv, -lnv)

SYNOPSIS

#include <sys/nv.h>

*nvlist_t **
nvlist_create(*int flags*);

void
nvlist_destroy(*nvlist_t *nvl*);

int
nvlist_error(*const nvlist_t *nvl*);

void
nvlist_set_error(*nvlist_t *nvl, int error*);

bool
nvlist_empty(*const nvlist_t *nvl*);

int
nvlist_flags(*const nvlist_t *nvl*);

bool
nvlist_in_array(*const nvlist_t *nvl*);

*nvlist_t **
nvlist_clone(*const nvlist_t *nvl*);

void
nvlist_dump(*const nvlist_t *nvl, int fd*);

void

nvlist_fdump(*const nvlist_t *nvl, FILE *fp*);

size_t
nvlist_size(*const nvlist_t *nvl*);

*void **
nvlist_pack(*const nvlist_t *nvl, size_t *sizep*);

*nvlist_t **
nvlist_unpack(*const void *buf, size_t size, int flags*);

int
nvlist_send(*int sock, const nvlist_t *nvl*);

*nvlist_t **
nvlist_recv(*int sock, int flags*);

*nvlist_t **
nvlist_xfer(*int sock, nvlist_t *nvl, int flags*);

*const char **
nvlist_next(*const nvlist_t *nvl, int *typep, void **cookiep*);

bool
nvlist_exists(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_type(*const nvlist_t *nvl, const char *name, int type*);

bool
nvlist_exists_null(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_bool(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_number(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_string(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_nvlist(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_descriptor(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_binary(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_bool_array(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_number_array(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_string_array(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_nvlist_array(*const nvlist_t *nvl, const char *name*);

bool
nvlist_exists_descriptor_array(*const nvlist_t *nvl, const char *name*);

void
nvlist_add_null(*nvlist_t *nvl, const char *name*);

void
nvlist_add_bool(*nvlist_t *nvl, const char *name, bool value*);

void
nvlist_add_number(*nvlist_t *nvl, const char *name, uint64_t value*);

void
nvlist_add_string(*nvlist_t *nvl, const char *name, const char *value*);

void
nvlist_add_stringf(*nvlist_t *nvl, const char *name, const char *valuefmt, ...*);

void

nvlist_add_stringv(*nvlist_t *nvl, const char *name, const char *valuefmt, va_list valueap*);

void
nvlist_add_nvlist(*nvlist_t *nvl, const char *name, const nvlist_t *value*);

void
nvlist_add_descriptor(*nvlist_t *nvl, const char *name, int value*);

void
nvlist_add_binary(*nvlist_t *nvl, const char *name, const void *value, size_t size*);

void
nvlist_add_bool_array(*nvlist_t *nvl, const char *name, const bool *value, size_t nitems*);

void
nvlist_add_number_array(*nvlist_t *nvl, const char *name, const uint64_t *value, size_t nitems*);

void
nvlist_add_string_array(*nvlist_t *nvl, const char *name, const char * const * value, size_t nitems*);

void
nvlist_add_nvlist_array(*nvlist_t *nvl, const char *name, const nvlist_t * const * value, size_t nitems*);

void
nvlist_add_descriptor_array(*nvlist_t *nvl, const char *name, const int *value, size_t nitems*);

void
nvlist_move_string(*nvlist_t *nvl, const char *name, char *value*);

void
nvlist_move_nvlist(*nvlist_t *nvl, const char *name, nvlist_t *value*);

void
nvlist_move_descriptor(*nvlist_t *nvl, const char *name, int value*);

void
nvlist_move_binary(*nvlist_t *nvl, const char *name, void *value, size_t size*);

void
nvlist_move_bool_array(*nvlist_t *nvl, const char *name, bool *value, size_t nitems*);

void
nvlist_move_number_array(*nvlist_t *nvl, const char *name, uint64_t *value, size_t nitems*);

void
nvlist_move_string_array(*nvlist_t *nvl, const char *name, char **value, size_t nitems*);

void
nvlist_move_nvlist_array(*nvlist_t *nvl, const char *name, nvlist_t **value, size_t nitems*);

void
nvlist_move_descriptor_array(*nvlist_t *nvl, const char *name, int *value, size_t nitems*);

bool
nvlist_get_bool(*const nvlist_t *nvl, const char *name*);

uint64_t
nvlist_get_number(*const nvlist_t *nvl, const char *name*);

*const char **
nvlist_get_string(*const nvlist_t *nvl, const char *name*);

*const nvlist_t **
nvlist_get_nvlist(*const nvlist_t *nvl, const char *name*);

int
nvlist_get_descriptor(*const nvlist_t *nvl, const char *name*);

*const void **
nvlist_get_binary(*const nvlist_t *nvl, const char *name, size_t *sizep*);

*const bool **
nvlist_get_bool_array(*const nvlist_t *nvl, const char *name, size_t *nitems*);

*const uint64_t **
nvlist_get_number_array(*const nvlist_t *nvl, const char *name, size_t *nitems*);

*const char * const **
nvlist_get_string_array(*const nvlist_t *nvl, const char *name, size_t *nitems*);

*const nvlist_t * const **

nvlist_get_nvlist_array(*const nvlist_t *nvl, const char *name, size_t *nitems*);

*const int **
nvlist_get_descriptor_array(*const nvlist_t *nvl, const char *name, size_t *nitems*);

*const nvlist_t **
nvlist_get_parent(*const nvlist_t *nvl, void **cookiep*);

*const nvlist_t **
nvlist_get_array_next(*const nvlist_t *nvl*);

*const nvlist_t **
nvlist_get_pararr(*const nvlist_t *nvl, void **cookiep*);

bool
nvlist_take_bool(*nvlist_t *nvl, const char *name*);

uint64_t
nvlist_take_number(*nvlist_t *nvl, const char *name*);

*char **
nvlist_take_string(*nvlist_t *nvl, const char *name*);

*nvlist_t **
nvlist_take_nvlist(*nvlist_t *nvl, const char *name*);

int
nvlist_take_descriptor(*nvlist_t *nvl, const char *name*);

*void **
nvlist_take_binary(*nvlist_t *nvl, const char *name, size_t *sizep*);

*bool **
nvlist_take_bool_array(*nvlist_t *nvl, const char *name, size_t *nitems*);

*uint64_t ***
nvlist_take_number_array(*nvlist_t *nvl, const char *name, size_t *nitems*);

*char ***
nvlist_take_string_array(*nvlist_t *nvl, const char *name, size_t *nitems*);

*nvlist_t **
nvlist_take_nvlist_array(*nvlist_t *nvl, const char *name, size_t *nitems*);

*int **
nvlist_take_descriptor_array(*nvlist_t *nvl, const char *name, size_t *nitems*);

void
nvlist_append_bool_array(*nvlist_t *nvl, const char *name, const bool value*);

void
nvlist_append_number_array(*nvlist_t *nvl, const char *name, const uint64_t value*);

void
nvlist_append_string_array(*nvlist_t *nvl, const char *name, const char * const value*);

void
nvlist_append_nvlist_array(*nvlist_t *nvl, const char *name, const nvlist_t * const value*);

void
nvlist_append_descriptor_array(*nvlist_t *nvl, const char *name, int value*);

void
nvlist_free(*nvlist_t *nvl, const char *name*);

void
nvlist_free_type(*nvlist_t *nvl, const char *name, int type*);

void
nvlist_free_null(*nvlist_t *nvl, const char *name*);

void
nvlist_free_bool(*nvlist_t *nvl, const char *name*);

void
nvlist_free_number(*nvlist_t *nvl, const char *name*);

void
nvlist_free_string(*nvlist_t *nvl, const char *name*);

void

nvlist_free_nvlist(*nvlist_t *nvl, const char *name*);

void
nvlist_free_descriptor(*nvlist_t *nvl, const char *name*);

void
nvlist_free_binary(*nvlist_t *nvl, const char *name*);

void
nvlist_free_bool_array(*nvlist_t *nvl, const char *name*);

void
nvlist_free_number_array(*nvlist_t *nvl, const char *name*);

void
nvlist_free_string_array(*nvlist_t *nvl, const char *name*);

void
nvlist_free_nvlist_array(*nvlist_t *nvl, const char *name*);

void
nvlist_free_descriptor_array(*nvlist_t *nvl, const char *name*);

DESCRIPTION

The **libnv** library allows to easily manage name value pairs as well as send and receive them over sockets. A group (list) of name value pairs is called an **nvlist**. The API supports the following data types:

null (**NV_TYPE_NULL**)
There is no data associated with the name.

bool (**NV_TYPE_BOOL**)
The value can be either true or false.

number (**NV_TYPE_NUMBER**)
The value is a number stored as *uint64_t*.

string (**NV_TYPE_STRING**)
The value is a C string.

nvlist (**NV_TYPE_NVLIST**)
The value is a nested nvlist.

descriptor (**NV_TYPE_DESCRIPTOR**)
The value is a file descriptor. Note that file descriptors can be sent only over unix(4) domain sockets.

binary (**NV_TYPE_BINARY**)
The value is a binary buffer.

bool array (**NV_TYPE_BOOL_ARRAY**)
The value is an array of boolean values.

number array (**NV_TYPE_NUMBER_ARRAY**)
The value is an array of numbers, each stored as *uint64_t*.

string array (**NV_TYPE_STRING_ARRAY**)
The value is an array of C strings.

nvlist array (**NV_TYPE_NVLIST_ARRAY**)
The value is an array of nvlists. When an nvlist is added to an array, it becomes part of the primary nvlist. Traversing these arrays can be done using the **nvlist_get_array_next**() and **nvlist_get_pararr**() functions.

descriptor array (**NV_TYPE_DESCRIPTOR_ARRAY**)
The value is an array of files descriptors.

The **nvlist_create**() function allocates memory and initializes an nvlist.

The following flag can be provided:

NV_FLAG_IGNORE_CASE Perform case-insensitive lookups of provided names.
NV_FLAG_NO_UNIQUE Names in the nvlist do not have to be unique.

The **nvlist_destroy**() function destroys the given nvlist. Function does nothing if NULL nvlist is provided. Function never modifies the *errno* global variable.

The **nvlist_error**() function returns any error value that the nvlist accumulated. If the given nvlist is NULL the ENOMEM error will be returned.

The **nvlist_set_error**() function sets an nvlist to be in the error state. Subsequent calls to **nvlist_error**() will return the given error value. This function cannot be used to clear the error state from an nvlist. This function does nothing if the nvlist is already in the error state.

The **nvlist_empty**() function returns true if the given nvlist is empty and false otherwise. The nvlist must not be in error state.

The **nvlist_flags**() function returns flags used to create the nvlist with the **nvlist_create**() function.

The **nvlist_in_array**() function returns true if *nvl* is part of an array that is a member of another nvlist.

The **nvlist_clone**() functions clones the given nvlist. The clone shares no resources with its origin. This also means that all file descriptors that are part of the nvlist will be duplicated with the dup(2) system call before placing them in the clone.

The **nvlist_dump**() dumps nvlist content for debugging purposes to the given file descriptor *fd*.

The **nvlist_fdump**() dumps nvlist content for debugging purposes to the given file stream *fp*.

The **nvlist_size**() function returns the size of the given nvlist after converting it to binary buffer with the **nvlist_pack**() function.

The **nvlist_pack**() function converts the given nvlist to a binary buffer. The function allocates memory for the buffer, which should be freed with the free(3) function. If the *sizep* argument is not NULL, the size of the buffer will be stored there. The function returns NULL in case of an error (allocation failure). If the nvlist contains any file descriptors NULL will be returned. The nvlist must not be in error state.

The **nvlist_unpack**() function converts the given buffer to the nvlist. The *flags* argument defines what type of the top level nvlist is expected to be. Flags are set up using the **nvlist_create**() function. If the nvlist flags do not match the flags passed to **nvlist_unpack**(), the nvlist will not be returned. Every nested nvlist list should be checked using **nvlist_flags**() function. The function returns NULL in case of an error.

The **nvlist_send**() function sends the given nvlist over the socket given by the *sock* argument. Note that nvlist that contains file descriptors can only be send over unix(4) domain sockets.

The **nvlist_recv**() function receives nvlist over the socket given by the *sock* argument. The *flags* argument defines what type of the top level nvlist is expected to be. Flags are set up using the **nvlist_create**() function. If the nvlist flags do not match the flags passed to **nvlist_recv**(), the nvlist will not be returned. Every nested nvlist list should be checked using **nvlist_flags**() function.

The **nvlist_xfer**() function sends the given nvlist over the socket given by the *sock* argument and receives nvlist over the same socket. The *flags* argument defines what type of the top level nvlist is expected to be. Flags are set up using the **nvlist_create**() function. If the nvlist flags do not match the flags passed to **nvlist_xfer**(), the nvlist will not be returned. Every nested nvlist list should be checked using **nvlist_flags**() function. The given nvlist is always destroyed.

The **nvlist_next**() function iterates over the given nvlist returning names and types of subsequent elements. The *cookiep* argument allows the function to figure out which element should be returned next. The **cookiep* should be set to NULL for the first call and should not be changed later. Returning NULL means there are no more elements on the nvlist. The *typep* argument can be NULL. Elements may not be removed from the nvlist while traversing it. The nvlist must not be in error state. Note that **nvlist_next**() will handle *cookiep* being set to NULL. In this case first element is returned or NULL if nvlist is empty. This behavior simplifies removing the first element from the list.

The **nvlist_exists**() function returns true if element of the given name exists (besides of its type) or false otherwise. The nvlist must not be in error state.

The **nvlist_exists_type**() function returns true if element of the given name and the given type exists or false otherwise. The nvlist must not be in error state.

The **nvlist_exists_null**(), **nvlist_exists_bool**(), **nvlist_exists_number**(), **nvlist_exists_string**(), **nvlist_exists_nvlist**(), **nvlist_exists_descriptor**(), **nvlist_exists_binary**(), **nvlist_exists_bool_array**(), **nvlist_exists_number_array**(), **nvlist_exists_string_array**(), **nvlist_exists_nvlist_array**(), **nvlist_exists_descriptor_array**() functions return true if element of the given name and the given type determined by the function name exists or false otherwise. The nvlist must not be in error state.

The **nvlist_add_null**(), **nvlist_add_bool**(), **nvlist_add_number**(), **nvlist_add_string**(), **nvlist_add_stringf**(), **nvlist_add_stringv**(), **nvlist_add_nvlist**(), **nvlist_add_descriptor**(), **nvlist_add_binary**(), **nvlist_add_bool_array**(), **nvlist_add_number_array**(), **nvlist_add_string_array**(), **nvlist_add_nvlist_array**(), **nvlist_add_descriptor_array**() functions add element to the given nvlist. When adding string or binary buffer the functions will allocate memory and copy the data over. When adding nvlist, the nvlist will be cloned and clone will be added. When adding descriptor, the descriptor will be duplicated using the dup(2) system call and the new descriptor will be added. The array functions will fail if there are any NULL elements in the array, or if the array pointer is NULL. If an error occurs while adding new element, internal error is set which can be examined using the **nvlist_error**() function.

The **nvlist_move_string**(), **nvlist_move_nvlist**(), **nvlist_move_descriptor**(), **nvlist_move_binary**(), **nvlist_move_bool_array**(), **nvlist_move_number_array**(), **nvlist_move_string_array**(), **nvlist_move_nvlist_array**(), **nvlist_move_descriptor_array**() functions add new element to the given

nvlist, but unlike **nvlist_add_<type>**() functions they will consume the given resource. In the case of strings, descriptors, or nvlists every elements must be unique, or it could cause a double free. The array functions will fail if there are any NULL elements, or if the array pointer is NULL. If an error occurs while adding new element, the resource is destroyed and internal error is set which can be examined using the **nvlist_error**() function.

The **nvlist_get_bool**(), **nvlist_get_number**(), **nvlist_get_string**(), **nvlist_get_nvlist**(), **nvlist_get_descriptor**(), **nvlist_get_binary**(), **nvlist_get_bool_array**(), **nvlist_get_number_array**(), **nvlist_get_string_array**(), **nvlist_get_nvlist_array**(), **nvlist_get_descriptor_array**() functions return the value that corresponds to the given key name. In the case of strings, nvlists, descriptors, binary, or arrays, the returned resource should not be modified - they still belong to the nvlist. If an element of the given name does not exist, the program will be aborted. To avoid this, the caller should check for the existence of the name before trying to obtain the value, or use the dnvlist(3) extension, which can provide a default value in the case of a missing element. The nvlist must not be in error state.

The **nvlist_get_parent**() function returns the parent nvlist of the nested nvlist.

The **nvlist_get_array_next**() function returns the next element from the array or NULL if the nvlist is not in array or it is the last element. Note that **nvlist_get_array_next**() only works if you added the nvlist array using the **nvlist_move_nvlist_array**() or **nvlist_add_nvlist_array**() functions.

The **nvlist_get_pararr**() function returns the next element in the array, or if not available the parent of the nested nvlist.

The **nvlist_take_bool**(), **nvlist_take_number**(), **nvlist_take_string**(), **nvlist_take_nvlist**(), **nvlist_take_descriptor**(), **nvlist_take_binary**(), **nvlist_take_bool_array**(), **nvlist_take_number_array**(), **nvlist_take_string_array**(), **nvlist_take_nvlist_array**(), **nvlist_take_descriptor_array**() functions return value associated with the given name and remove the element from the nvlist. In case of string and binary values, the caller is responsible for free returned memory using the free(3) function. In case of nvlist, the caller is responsible for destroying returned nvlist using the **nvlist_destroy**() function. In case of descriptor, the caller is responsible for closing returned descriptor using the **close**(*2*) system call. If an element of the given name does not exist, the program will be aborted. To avoid that the caller should check for the existence of the given name before trying to obtain the value, or use the dnvlist(3) extension, which can provide a default value in the case of a missing element. In the case of an array of strings or binary values, the caller is responsible for freeing every element of the array using the free(3) function. In the case of an array of nvlists, the caller is responsible for destroying every element of array using the **nvlist_destroy**() function. In the case of descriptors, the caller is responsible for closing every element of array using the **close**(*2*) system call. In every case involving an array, the caller must also free the pointer to the array using the free(3) function. The nvlist must not be in error state.

The **nvlist_append_bool_array**(), **nvlist_append_number_array**(), **nvlist_append_string_array**(), **nvlist_append_nvlist_array**(), **nvlist_append_descriptor_array**() functions append an element to the existing array using the same semantics as the add functions (i.e. the element will be copied when applicable). If the array for a given key does not exist, then it will be created as if using the **nvlist_add_<type>_array**() function. The internal error is set on append failure.

The **nvlist_free**() function removes element of the given name from the nvlist (besides of its type) and frees all resources associated with it. If element of the given name does not exist, the program will be aborted. The nvlist must not be in error state.

The **nvlist_free_type**() function removes element of the given name and the given type from the nvlist and frees all resources associated with it. If element of the given name and the given type does not exist, the program will be aborted. The nvlist must not be in error state.

The **nvlist_free_null**(), **nvlist_free_bool**(), **nvlist_free_number**(), **nvlist_free_string**(), **nvlist_free_nvlist**(), **nvlist_free_descriptor**(), **nvlist_free_binary**(), **nvlist_free_bool_array**(), **nvlist_free_number_array**(), **nvlist_free_string_array**(), **nvlist_free_nvlist_array**(), **nvlist_free_descriptor_array**() functions remove element of the given name and the given type determined by the function name from the nvlist and free all resources associated with it. If element of the given name and the given type does not exist, the program will be aborted. The nvlist must not be in error state.

NOTES

The **nvlist_pack**() and **nvlist_unpack**() functions handle the byte-order conversions, so the binary buffer can be packed/unpacked by the hosts with the different endianness.

EXAMPLES

The following example demonstrates how to prepare an nvlist and send it over unix(4) domain socket.

```
nvlist_t *nvl;
int fd;

fd = open("/tmp/foo", O_RDONLY);
if (fd < 0)
        err(1, "open(\"/tmp/foo\") failed");

nvl = nvlist_create(0);
/*
 * There is no need to check if nvlist_create() succeeded,
 * as the nvlist_add_<type>() functions can cope.
```

```
 * If it failed, nvlist_send() will fail.
 */
nvlist_add_string(nvl, "filename", "/tmp/foo");
nvlist_add_number(nvl, "flags", O_RDONLY);
/*
 * We just want to send the descriptor, so we can give it
 * for the nvlist to consume (that’s why we use nvlist_move
 * not nvlist_add).
 */
nvlist_move_descriptor(nvl, "fd", fd);
if (nvlist_send(sock, nvl) < 0) {
        nvlist_destroy(nvl);
        err(1, "nvlist_send() failed");
}
nvlist_destroy(nvl);
```

Receiving nvlist and getting data:

```
nvlist_t *nvl;
const char *command;
char *filename;
int fd;

nvl = nvlist_recv(sock, 0);
if (nvl == NULL)
        err(1, "nvlist_recv() failed");

/* For command we take pointer to nvlist’s buffer. */
command = nvlist_get_string(nvl, "command");
/*
 * For filename we remove it from the nvlist and take
 * ownership of the buffer.
 */
filename = nvlist_take_string(nvl, "filename");
/* The same for the descriptor. */
fd = nvlist_take_descriptor(nvl, "fd");

printf("command=%s filename=%s fd=%d0, command, filename, fd);

nvlist_destroy(nvl);
```

```
free(filename);
close(fd);
/* command was freed by nvlist_destroy() */
```

Iterating over nvlist:

```
nvlist_t *nvl;
const char *name;
void *cookie;
int type;

nvl = nvlist_recv(sock, 0);
if (nvl == NULL)
        err(1, "nvlist_recv() failed");

cookie = NULL;
while ((name = nvlist_next(nvl, &type, &cookie)) != NULL) {
        printf("%s=", name);
        switch (type) {
        case NV_TYPE_NUMBER:
                printf("%ju", (uintmax_t)nvlist_get_number(nvl, name));
                break;
        case NV_TYPE_STRING:
                printf("%s", nvlist_get_string(nvl, name));
                break;
        default:
                printf("N/A");
                break;
        }
        printf("\n");
}
```

Iterating over every nested nvlist:

```
nvlist_t *nvl;
const char *name;
void *cookie;
int type;

nvl = nvlist_recv(sock, 0);
```

```
if (nvl == NULL)
        err(1, "nvlist_recv() failed");

cookie = NULL;
do {
        while ((name = nvlist_next(nvl, &type, &cookie)) != NULL) {
                if (type == NV_TYPE_NVLIST) {
                        nvl = nvlist_get_nvlist(nvl, name);
                        cookie = NULL;
                }
        }
} while ((nvl = nvlist_get_parent(nvl, &cookie)) != NULL);
```

Iterating over every nested nvlist and every nvlist element:

```
nvlist_t *nvl;
const nvlist_t * const *array;
const char *name;
void *cookie;
int type;

nvl = nvlist_recv(sock, 0);
if (nvl == null)
        err(1, "nvlist_recv() failed");

cookie = null;
do {
        while ((name = nvlist_next(nvl, &type, &cookie)) != NULL) {
                if (type == NV_TYPE_NVLIST) {
                        nvl = nvlist_get_nvlist(nvl, name);
                        cookie = NULL;
                } else if (type == NV_TYPE_NVLIST_ARRAY) {
                        nvl = nvlist_get_nvlist_array(nvl, name, NULL)[0];
                        cookie = NULL;
                }
        }
} while ((nvl = nvlist_get_pararr(nvl, &cookie)) != NULL);
```

Or alternatively:

```
nvlist_t *nvl, *tmp;
const nvlist_t * const *array;
const char *name;
void *cookie;
int type;

nvl = nvlist_recv(sock, 0);
if (nvl == null)
        err(1, "nvlist_recv() failed");

cooke = NULL;
tmp = nvl;
do {
        do {
                nvl = tmp;
                while ((name = nvlist_next(nvl, &type, &cookie)) != NULL) {
                        if (type == NV_TYPE_NVLIST) {
                                nvl = nvlist_get_nvlist(nvl, name);
                                cookie = NULL;
                        } else if (type == NV_TYPE_NVLIST_ARRAY) {
                                nvl = nvlist_get_nvlist_array(nvl, name,
                                   NULL)[0];
                                cookie = NULL;
                        }
                }
                cookie = NULL;
        } while ((tmp = nvlist_get_array_next(nvl)) != NULL);
} while ((tmp = nvlist_get_parent(nvl, &cookie)) != NULL);
```

SEE ALSO

close(2), dup(2), open(2), err(3), free(3), printf(3), unix(4)

HISTORY

The **libnv** library appeared in FreeBSD 11.0.

AUTHORS

The **libnv** library was implemented by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation.