NAME

vnaproperty_vtype, vnaproperty_vcount, vnaproperty_vkeys, vnaproperty_vget, vnaproperty_vset, vnaproperty_vdelete, vnaproperty_vget_subtree, vnaproperty_vset_subtree, vnaproperty_type, vnaproperty_count, vnaproperty_keys, vnaproperty_get, vnaproperty_set, vnaproperty_delete, vnaproperty_get_subtree, vnaproperty_set_subtree, vnaproperty_copy, vnaproperty_quote_key, vnaproperty_import_yaml_from_string, vnaproperty_import_yaml_from_file, vnaproperty_export_yaml_to_file – VNA YAML interface

SYNOPSIS

#include <vnaproperty.h>

int vnaproperty_type(const vnaproperty_t **root*, const char ** *format*, ...);

int vnaproperty_count(const vnaproperty_t **root*, const char ** *format*, ...);

const char **vnaproperty_keys(const vnaproperty_t **root*, const char ** *format*, ...);

const char *vnaproperty_get(const vnaproperty_t **root*, const char ** *format*, ...);

int vnaproperty_set(vnaproperty_t ***rootptr*, const char ** *format*, ...);

int vnaproperty_delete(vnaproperty_t ***rootptr*, const char ** *format*, ...);

vnaproperty_t *vnaproperty_get_subtree(const vnaproperty_t **root*, const char ** *format*, ...);

vnaproperty_t **vnaproperty_set_subtree(vnaproperty_t ***rootptr*, const char ** *format*, ...);

Variants Taking a va_list

int vnaproperty_vtype(const vnaproperty_t **root*, const char ** *format*, va_list *ap*);

int vnaproperty_vcount(const vnaproperty_t **root*, const char ** *format*, va_list *ap*);

const char **vnaproperty_vkeys(const vnaproperty_t **root*, const char ** *format*, va_list *ap*);

const char *vnaproperty_vget(const vnaproperty_t **root*, const char ** *format*, va_list *ap*);

int vnaproperty_vset(vnaproperty_t ***rootptr*, const char ** *format*, va_list *ap*);

int vnaproperty_vdelete(vnaproperty_t ***rootptr*, const char ** *format*, va_list *ap*);

vnaproperty_t *vnaproperty_vget_subtree(const vnaproperty_t **root*,
const char ** *format*, va_list *ap*);

vnaproperty_t **vnaproperty_vset_subtree(vnaproperty_t ***rootptr*,
const char ** *format*, va_list *ap*);

Copy, Import, Export, Misc.

char *vnaproperty_quote_key(const char ***key*);

int vnaproperty_copy(vnaproperty_t ***destination*, const vnaproperty_t ***source*);

int vnaproperty_import_yaml_from_string(vnaproperty_t ***rootptr*, const char ***input*,
vnaerr_error_fn_t ***error_fn*, void ***error_arg*);

int vnaproperty_import_yaml_from_file(vnaproperty_t ***rootptr*, FILE ** *fp*, const char ** *filename*,
vnaerr_error_fn_t ***error_fn*, void ***error_arg*);

int vnaproperty_export_yaml_to_file(const vnaproperty_t **root*, FILE ** *fp*, const char ** *filename*,
vnaerr_error_fn_t ***error_fn*, void ***error_arg*);

Link with *-lvna -lyaml -lm*.

DESCRIPTION

These functions provide a convenient interface for storing arbitrary data (properties) as key-value pairs and lists of values. The values themselves can be maps of key-value pairs and lists, making it possible to organize data hierarchically as a tree structure. Originally, the vnaproperty interface was used to record VNA configuration and other conditions under which a calibration was made (see **vnacal**(3)), but the API has since been expanded to make it useful for other purposes. Property data can be exported to and imported

from files in YAML format.

Most of the functions take a format string and variable arguments as in **sprintf**(3). While all scalars are stored internally as text strings, the sprint-like interface makes it easy to store numeric values without first having to convert them. The interface also makes substitutions on the left-hand side possible. Example:

```
vnaproperty_t *root = NULL;
int n_ports = 2;
int my_index = 3;

/* Create a map and set the VNA_Model key to a fixed value. */
vnaproperty_set(&root, "VNA_Model=ACME 1050");

/* Set VNA_ports to the value in the n_ports variable */
vnaproperty_set(&root, "VNA_ports=%d", n_ports);

/* Set the key name formed from the value in my_index to pi. */
vnaproperty_set(&root, "my_property%d=%f", my_index, M_PI);

/* Retrieve and print the values set above. */
printf("%s\n", vnaproperty_get(root, "VNA_Model"));
printf("%s\n", vnaproperty_get(root, "VNA_ports"));
printf("%s\n", vnaproperty_get(root, "my_property%d", my_index));
```

Syntax of the Descriptor

In the **vnaproperty_set**() examples above, the string appearing left of the equals sign after %-expansion is called the descriptor. This section describes the descriptor syntax.

The descriptor may optionally begin with a dot. A descriptor consisting only of a dot represents the root of the tree.

Following the optional leading dot is a list of zero or more dot-separated map keys and list subscripts, describing a path down the tree. A few examples are: “abc”, “abc.def”, “[0]”, “names[0]”, “names[1]”, and “.abc.def[2][0].ghi”. The descriptor cannot be empty.

Map keys begin with a letter, underscore, UTF-8 encoded character or backslash-quoted character, followed by any number of letters, underscores, UTF-8 encoded characters, backslash-quoted characters, digits and minuses. Keys may contain multiple words separated by spaces; spaces between words do not need to be quoted. Keys are separated by dots.

List subscripts consist of a non-negative integer, a non-negative integer followed by a plus sign, or just a plus sign, enclosed within square brackets. An simple integer subscript indexes list items, with the first item having index zero, e.g. “[0]”. In contexts that modify the property tree, an integer subscript followed by a plus sign, e.g. “[5+]”, causes an item to be inserted into the list at the given position, moving existing elements up. A subscript consisting of a plus alone, e.g. “[+]” causes a new element to be appended to the end of the list. More than one subscript with a + can appear in the same descriptor string.

The descriptor may optionally end with empty braces (“{}”), empty brackets (“[]”), or if preceded by a key or subscript, a dot (“.”). The {} and [] suffixes require the last element in the path to match a map or list, respectively. A trailing dot represents the element of a map or list as opposed to the map or list entry itself: this distinction matters only in **vnaproperty_delete**(). For example, deleting “foo” removes foo from the map, but deleting “foo.” retains foo as a key, replacing its descendents with an empty subtree.

Functions

The **vnaproperty_type**() function returns the type of the specified element in the property tree. If returns ’m’ if the element is a map, ’l’ if it’s a list, ’s’ if it’s a scalar or -1 if the descriptor does not refer to a valid element.

The **vnaproperty_count**() function returns the number of elements in the given map or list. If the

descriptor doesn't refer to a map or list, the function fails with -1.

The **vnaproperty_keys**() function returns a vector of pointers to the keys of the specified map. The caller is responsible for freeing the memory of the returned vector but not the strings it points to, by a call to **free**(3). If the descriptor doesn't refer to a map, or if the function is unable to allocate memory for the vector, the function indicates failure by a return of NULL.

The **vnaproperty_get**() function returns the value of the specified scalar. If the descriptor doesn't refer to a scalar, the function returns NULL. Do not call **free**(3) on the returned string!

The **vnaproperty_set**() function places a scalar value into in the property tree, creating and replacing objects along the path as needed to make them conform to the descriptor string. Normally, the argument to this function has the form *descriptor*=*value*, where everything to the right of the equal sign is taken literally as the value to set. As a special-case, however, if the function is invoked with an argument of the form *descriptor*#, it doesn't place a scalar into the tree, but rather places a null value, e.g. ~ in YAML. You cannot create a null value using the first form; for example, the assignment *descriptor*=~ sets the descriptor literally to the string "~".

If an object along the given path exists but is of the wrong type, **vnapropert_set**() removes the conflicting element and all its descendents and replaces it with the desired element.

The **vnaproperty_delete**() function removes objects from the tree. With a descriptor of ".", it removes all elements of the tree and sets the root pointer to NULL. If the top-level of the tree is a map and the descriptor is "foo", **vnaproperty_delete**() removes foo from the map. If the top-level of the tree is a list and the descriptor is "[5]", **vnaproperty_delete**() removes the element at index 5, moving any elements with higher indices down by one. The behavior changes if the descriptor ends in dot. e.g. in the map example, if the descriptor is "foo.", foo is left in the map but its value is set to null. Simlarly, in the list example, if the descriptor is "[5].", then the element at index 5 is replaced with a null value, retaining its position. These differ from the #-form of **vnaproperty_set**() in that **vnaproperty_delete**() does not create or replace elements along the path that are inconsistent with the descriptor, but fails instead. For example, if the top level of the tree is a list but you call **vnaproperty_delete**() with a descriptor of "foo", it fails with a return of -1, whereas **vnaproperty_set**() would replace the entire list with a map and insert foo with a null value.

The **vnaproperty_get_subtree**() function returns the root of the subtree described by the descriptor. If the descriptor refers to nonexistent elements or is inconsistent with the tree, it fails with NULL. This function is useful for factoring out common code and reducing the length of the path that has to be traversed on each call.

Note that NULL is a valid return value if the subtree is empty. Setting **errno** to 0 before the call and checking if it's still zero afterwards can be used to distinguish an empty subtree from an error.

The **vnaproperty_set_subtree**() function is similar to **vnaproperty_get_subtree**() except that it creates and replaces objects along the given path, forcing the property tree to conform to the descriptor as in **vnaproperty_set**(), and instead of returning the root of the subtree, it returns the address of the root of the subtree, making it possible to subsequently use any of the modifying functions on the subtree.

The **vnaproperty_vtype**(), **vnaproperty_vcount**(), **vnaproperty_vget**(), **vnaproperty_vset**(), **vnaproperty_vdelete**(), **vnaproperty_vget_subtree**() and **vnaproperty_vset_subtree**() functions are the same as their non-v counterparts above, except that they are called with a **va_list** instead of a variable number of arguments. These functions do not call the **va_end**() macro. Because they invoke the **va_arg**() macro, the value of *ap* is undefined after the call. See **stdarg**(3).

The **vnaproperty_quote_key**() function returns a copy of *key* with backslash quotes inserted as needed to make it suitable for use in the descriptor string of the other functions. For example, suppose the top level of the tree is a map and we want to look-up *key* in the map. Further suppose that *key* may contain literal dots, e.g. "my.key". If we hand *key* to **vnaproperty_get**() directly, the dots will be interpreted as separators, searching first in our example for the key "my", then expecting the value to be a map containing "key". We can avoid the special meaning of dots, brackets and braces appearing in the key by calling **vnaproperty_quote_key**(). In our example, it returns "my\.key", making the dot literal. Example:

```
char *quoted_key;
const char *value;

if ((quoted_key = vnaproperty_quote_key(key)) == NULL) {
   ....handle error....
}
if ((value = vnaproperty_get("%s", quoted_key)) == NULL) {
   ....handle key does not exist....
}
free((void *)quoted_key); // finished with quoted_key
printf("value is: %s0, value);
```

A common use of **vnaproperty_quote_key**() is with **vnaproperty_keys**(), where we use the later to get the set of (unquoted) keys from a map, iterate over the keys, and look up each value. The caller is responsible for freeing the memory of the returned string by a call to **free**(3).

The **vnaproperty_copy**() function creates a deep copy of the property tree in *source* and places it in *destination*, replacing any existing content in *destination*.

The **vnaproperty_import_yaml_from_string**() builds a property tree from the YAML document found in *input*, and places it at *rootptr*, replacing any existing content. If *errfn* is non-NULL, then errors found in the input document are reported by calling *errfn* with a single-line string describing the error. The *errarg* argument is passed through to the user's function. See **vnaerr**(3).

In YAML, map keys don't have to be scalars – they can be arbitrarily complex subtrees of maps and lists. This library, however, supports only scalar map keys. If **vnaproperty_import_yaml_from_string**() encounters a map item with a non-scalar key, it reports a warning through *errfn* and skips over the map entry.

The **vnaproperty_import_yaml_from_file**() reads the YAML document from file pointer *fp*, builds a property tree from it and places the tree at *rootptr*, replacing any previous content. The *filename* argument is used only in error messages and doesn't have to refer to an actual file. The other arguments are as in **vnaproperty_import_yaml_from_string**().

The **vnaproperty_export_yaml_to_file**() function creates a YAML document from *root* and writes it to the open file pointer *fp*. The other arguments are the same as in **vnaproperty_import_yaml_from_file**().

RETURN VALUE

The **vnaproperty_type**() and **vnaproperty_vtype**() functions return 'm' for map, 'l' for list, 's' for scalar, or -1 for error. The **vnaproperty_count**() and **vnaproperty_vcount**() functions return a count of objects or -1 on error. The **vnaproperty_keys**() and **vnaproperty_vkeys**() functions return a dynamically allocated vector of pointers to keys or NULL on error. The **vnaproperty_get**() and **vnaproperty_vget**() functions return a pointer to a string value or NULL on error. The other integer valued functions return 0 on success or -1 on error. The **vnaproperty_get_subtree**() and **vnaproperty_vget_subtree**() return the root of subtree on success or NULL on error; the **vnaproperty_set_subtree**() and **vnaproperty_vset_subtree**() functions return the address of the root of the given subtree on success or NULL on error. The **vnaproperty_quote_key**() function returns a dynamically allocated string on success or NULL on error.

ERRORS

EINVAL

This error is returned in each of the following cases. The descriptor string is not well formed. The **vnaproperty_count**() function was invoked on a scalar value. The **vnaproperty_get**() function was invoked on an object that's not a scalar. In a non-set function, the descriptor string contains a key or subscript, but the corresonding object in the property tree is not a map or list, respectively. In a non-set function, the descriptor string has {} or [], but the corresponding object in the property tree is not a map or list, respectively. In a non-set function, an insert [index+], or append [+] subscript was given.

ENOENT
This error is returned in each of the following cases. The given key doesn't exist in a map. The given subscript doesn't exist in a list.

ENOMEM
A function was unable to allocate memory.

NOTES

In YAML, it's possible to distinguish between numbers and strings, e.g. 3.14 is different from "3.14". In this library, the two are indistinguishable. Here, we consider a scalar to be a number if **strtol**(3) or **strtod**(3) is able to parse it.

EXAMPLES

The following example constructs a property tree from a YAML file and recursively prints it.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <vnaproperty.h>

/*
 * errfn: report YAML errors
 */
static void errfn(const char *message, void *error_arg,
        vnaerr_category_t category)
{
    fprintf(stderr, "%s\n", message);
}

/*
 * indent: indent level steps
 *   @level: number of indents
 */
static void indent(int level)
{
    for (int i = 0; i < level; ++i) {
        printf("  ");
    }
}

/*
 * print_subtree: print the subtree at root
 *   @root:  root of subtree to print
 *   @level: current indent level
 */
static void print_subtree(vnaproperty_t *root, int level)
{
    /*
     * Handle NULL subtree.
     */
    if (root == NULL) {
        printf("~");
        return;
    }
```

```
/*
 * Handle each node type...
 */
switch (vnaproperty_type(root, ".")) {
case 's': /* scalar */
    printf("\"%s\"", vnaproperty_get(root, "."));
    break;

case 'm': /* map */
    {
        const char **keys;

        /*
         * Get the list of keys.
         */
        if ((keys = vnaproperty_keys(root, "{}")) == NULL) {
            fprintf(stderr, "vnaproperty_keys: %s\n",
                    strerror(errno));
            exit(5);
        }

        /*
         * For each key, recurse.
         */
        printf("{\n");
        ++level;
        for (const char **cpp = keys; *cpp != NULL; ++cpp) {
            char *quoted;
            vnaproperty_t *subtree;

            if ((quoted = vnaproperty_quote_key(*cpp)) == NULL) {
                fprintf(stderr, "vnaproperty_quote_key: %s\n",
                        strerror(errno));
                exit(6);
            }
            subtree = vnaproperty_get_subtree(root, "%s", quoted);
            free((void *)quoted);
            indent(level);
            printf("%s: ", *cpp);
            print_subtree(subtree, level);
            if (cpp[1] != NULL) {
                printf(",");
            }
            printf("\n");
        }
        --level;
        indent(level);
        printf("}");
        free((void *)keys);
    }
    break;

case 'l': /* list */
    {
```

```
                int count;

                /*
                 * Get the count of elements in the list.
                 */
                if ((count = vnaproperty_count(root, "[]")) == -1) {
                    fprintf(stderr, "vnaproperty_count: %s\n",
                            strerror(errno));
                    exit(8);
                }

                /*
                 * For each element, recurse.
                 */
                printf("[\n");
                ++level;
                for (int i = 0; i < count; ++i) {
                    vnaproperty_t *subtree;

                    subtree = vnaproperty_get_subtree(root, "[%d]", i);
                    indent(level);
                    print_subtree(subtree, level);
                    if (i + 1 < count) {
                        printf(",");
                    }
                    printf("\n");
                }
                --level;
                indent(level);
                printf("]");
            }
            break;
        }
    }

    /*
     * main
     */
    int main(int argc, char **argv)
    {
        vnaproperty_t *root = NULL;
        FILE *fp;

        /*
         * Check usage.
         */
        if (argc != 2) {
            fprintf(stderr, "usage: yaml-file\n");
            exit(2);
        }

        /*
         * Build the property tree from the input file.
         */
```

```
        if ((fp = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "fopen: %s: %s\n", argv[1], strerror(errno));
            exit(3);
        }
        if (vnaproperty_import_yaml_from_file(&root, fp, argv[1],
                    errfn, NULL) == -1) {
            fprintf(stderr, "vnaproperty_import_yaml_from_file: %s: %s\n",
                    argv[1], strerror(errno));
            exit(4);
        }
        fclose(fp);

        /*
         * Print the tree.
         */
        print_subtree(root, 0);
        printf("\n");

        /*
         * Free all objects.
         */
        (void)vnaproperty_delete(&root, ".");

        exit(0);
    }
```

SEE ALSO

vnacal(3), **vnacal_new**(3), **vnaerr**(3)