Part IV

Programmer's Guide

Chapter 16

Programming the Directory

This part of the manual is written for implementors who wish to access the directory using the *libdsap*(3n) library. This might be an OSI application wishing to look up information, or an interactive directory user-interface. The interface to the directory is defined as a series of procedures which closely parallels the service defined in [CCI88]. This section defines the procedures, and shows how they relate to the standard. To avoid unnecessary wordiness, the user is expected to be familiar with the standard.

16.1 Conventions

The library defines various *C* structures, described in the following sections. Associated with the *C* structures, are routines which:

- allocate structures.
- free structures, including sub-structures.
- compare structures.
- copy structures.
- print structures.
- parse structures.

The following conventions are followed in many routines, which are then briefly identified later. For a general structure `x`, there will be the following routines:

- Allocation:

```
struct x *x_alloc ()
```

  Allocate memory for structure `x`.

- Freeing:

```
x_free (a)
struct x * a;
```

  Free structure `x`, and any nested structures.

  If the structure `x` is a linked-list type of structure, then as well as the routines described above, there will be routines of the form `x_comp_function` (where `function` is `alloc`, `free`, and so on). These act upon a single element of the list rather than the list as a whole.

- Copying:

```
struct x * x_cpy (a)
struct x *a;
```

  Return a copy of structure `a`.

- Comparing:

```
int     x_cmp (a, b)
struct x *a, *b;
```

  Return `0` if the structures `a` and `b` same. Return `-1` if `a<b`, or `1` if `a>b`. For some structures the notion of `a>b` and `a<b` is not defined, but will be consistent.

- Printing:

```
int     x_print (ps, a, format)
PS       ps;
struct x *a;
int      format;
```

Print the structure `a` to the presentation stream `ps`, using the specified format, either `READOUT`, for a pretty-printed format (for example the output of a showentry from DISH); or, `EDBOUT`, for a format suitable for inclusion in an EDB file. The `PS` structure is described in *Volume One* of this manual.

For the structures representing a DN and RDN there is an extra format: `DIROUT`, which is used to get a representation of the name suitable for inclusion in a UNIX filename (essentially `EDBOUT` with the `@` symbols replaced with `/` separators).

- String format:

```
struct x * str2x (str)
char * str;
```

Most attribute syntaxes have a string representation. A string of the required syntax can be converted to the relevant *C* structure with the "str2x" function. Each of the string parsers assumes the input string has had multiple spaces and tabs removed, and replaced with single spaces. The function

```
char * Tidy_String (a)
char * a;
```

will do the relevant tidying if required. Strings printed with the `EDBOUT` format of x_print should be parsable.

Note: `str2AttrV()` does not follow the "convention" and has an extra parameter, which is the expected syntax of the string.

The structures following the "convention" are listed below. The "prefix" column shows what value `x` should take to use the routines for the associated structure.

Structure	Prefix
RDN	rdn
DN	dn
AttributeType	AttrT
AttributeValue	AttrV
AV_Sequence	avs
Attr_Sequence	as
EntryInfo	entryinfo

16.2 Attributes

Attributes are fundamental components of the directory. Attributes have two parts, types and values, which are (respectfully) represented by two structures, `AttributeType` and `AttributeValue`.

To represent AttributeTypes, *libdsap*(3n) has a table (described in Section 14.11) which maps a string representation of attribute types onto OIDs.

This table must be loaded by calling

```
load_oid_table(name)
char * name;
```

The `name` parameter should be the name of the oidtable to load. For most purposes this should be "`oidtable`".

Before you load the table, any syntaxes you want special attribute handlers for should be loaded. To do this you will probably want to call

```
quipu_syntaxes()
```

which will load all the syntax handlers defined in Chapter 10. If not all there syntaxes are required, then a subset can be loaded by calling only the handlers required.

The call

```
dsap_init(argc,argv)
int *argc;
char *** argv;
```

can be used to load the oidtable for you, with the additional benifit that it will read the file */etc/dsaptailor* to find the location of the oidtable. The parameters it expects are the pointers to the `argc` and `argv` parameters passed to your "main()" routine. If a "`-c`" flag is found then this is used as the name of the DSA to contact as per DISH (see Section 4.1.13). A "`-t`" can be used to specify which *dsaptailor* file to use, and "`-T`" to define an oidtable. As with loading the tables, the syntaxes you require should have been loaded before you make a call to `dsa_init` (Beware: `dsap_init` removes flags it interprets from the parameter list, if you care, pass a copy of them, or pass nulls!).

The oidtables are represented using the following structures:

```
typedef struct {
        char    * ot_name;
        char    * ot_stroid;
        OID       ot_oid;
        OID       ot_aliasoid;
} oid_table;
#define NULLTABLE ((oid_table * )0)

typedef struct {
        oid_table oa_ot;
        short     oa_syntax;
} oid_table_attr, * AttributeType;
#define NULLTABLE_ATTR ((oid_table_attr *)0)
#define NULLAttrT ((oid_table_attr *)0)
```

The table consists of two fields, an integer syntax : `oa_syntax`. This is an integer handle onto the abstract syntax of the attribute value. The value "`0`" indicates an unknown syntax hence "ASN.1" is used.

The second field `oa_ot`, is a reference to an `oid_table` structure, which contains the oid, `ot_oid`; a numeric string representation of the oid, `ot_stroid`; and the local key string, `ot_name`.

The field `ot_aliasoid` is used to store an alternative OID for the same attribute. This is used to allow an attribute to change its OID over a period of time.

To create an `AttributeType` given a string representation of an OID, you should use

```
AttributeType str2AttrT (str)
char   *str;
```

which will look up the string supplied in the relevant OID tables and returns an `AttributeType` pointer into the oidtables.

There are various low level routines to manipulate entries in the OID table; however, the `AttributeType` routines described in Section 16.1 will be sufficient for most uses, so the details of these "invisible" routines are omitted here.

Attribute values are represented by the structure `AttributeValue`:

```
typedef struct attrVal {
        short      av_syntax;
        caddr_t    av_struct;
} * AttributeValue;

#define NULLAttrV (AttributeValue) NULL
```

The `AttributeValue` structure can hold attribute values in many formats. Different syntaxes use different specialised *C* structures pointed to by `av_struct`, the field `av_syntax` defines the syntax represented by the `av_struct` field. Unknown syntaxes are represented as a presentation element indicated by the syntax being zero. Section 16.4 describes this more fully.

Attribute values can now be linked together to form a multivalue attribute value. For this the `AV_Sequence` structure is used.

```
typedef struct avseqcomp {
    attrVal             avseq_av;
    struct avseqcomp    *avseq_next;
} avseqcomp, *AV_Sequence;

#define NULLAV ((AV_Sequence) 0)
```

This is simply a linked list of AttributeValues.

To create an `AV_Sequence` the two routines

```
AV_Sequence avs_comp_new (av)
AttributeValue av;
```

```
AV_Sequence avs_merge (a,b)
AV_Sequence a,b;
```

can be used.

Finally an attribute can now be formed by linking the attribute type, and multiple attribute values using the `AttrSequence` structure:

```
typedef struct attrcomp {
    AttributeType       attr_type;
    AV_Sequence         attr_value;
    struct attrcomp     *attr_link;
    struct acl_info     *attr_acl;
} attrcomp, *Attr_Sequence;

#define NULLATTR ((Attr_Sequence) 0)
```

This structure is used singularly to represent an attribute, and as a linked list (linked using `attr_link`) to represent a set of attributes.

To create an `Attr_Sequence` the two routines

```
Attr_Sequence as_comp_new (at,as,acl)
AttributeType at;
AV_Sequence as;
struct acl_info * acl;

Attr_Sequence as_merge (a,b)
Attr_Sequence a,b;
```

can be used. The `acl` parameter is only used by the DSA, for use in a DUA it should be set to `NULLACL_INFO`.

The routine

```
Attr_Sequence as_combine (as,str)
AV_Sequence as;
char * str;
```

is also provided, as an optimisation of

```
as_merge(as,str2as(str));
```

The approach to access control is defined in [Kil89b]. The representation of the attribute for access control is given by the structure shown below. This closely follows the ASN.1 definition.

```
struct acl_info {
    int             acl_categories;
#define ACL_NONE        0
#define ACL_DETECT      1
#define ACL_COMPARE     2
#define ACL_READ        3
#define ACL_ADD         4
#define ACL_WRITE       5
    int             acl_selector_type;
#define ACL_ENTRY       0
#define ACL_OTHER       1
#define ACL_PREFIX      2
#define ACL_GROUP       3
    DN              acl_name;
    struct acl_info *acl_next;
};
```

The DN `acl_name` is used for the type `ACL_PREFIX` and `ACL_OTHER` only.

16.3 Distinguished Names

The structures which define names are critical. Names are essentially made up of attributes, but because of their importance, a different — more specific structure is used.

```
typedef struct rdncomp {
    AttributeType       rdn_at;
    attrVal             rdn_av;
    struct rdncomp      *rdn_next;
} rdncomp, *RDN;

#define NULLRDN ((RDN) 0)
```

An RDN consists of a set of attribute types (`rdn_at`) and attribute values (`rdn_av`); the set is linked using `rdn_next`.

RDNs can the be joined in a sequence to form a distinguished name (DN):