17.7.1 Entry Information Selection

Operations that return an Entry Information structure, will have an "Entry Information Selection" in their arguments to specify exactly what the DSA should return. The structure is represented thus:

```
typedef struct {
    char                eis_allattributes;
    Attr_Sequence       eis_select;
    char                eis_infotypes;
#define EIS_ATTRIBUTETYPESONLY  0
#define EIS_ATTRIBUTESANDVALUES 1
} EntryInfoSelection;
```

The parameters are used as follows:

`eis_allattributes`: If `TRUE` the all attributes are required.

`eis_select`: If `eis_allattributes` is `FALSE` then this field is used to specify the attributes you want. *Only* the attribute types of this structure need to be set; any attribute values will be ignored.

`eis_infotypes`: If `EIS_ATTRIBUTETYPESONLY` is set, then only attribute types will be returned; otherwise the values will also be returned.

17.7.2 Entry Information

The structure "Entry Information" is used by some of the operations to return data to the DUA.

```
typedef struct entrystruct {
    DN                  ent_dn;
    Attr_Sequence       ent_attr;
    int                 ent_iscopy;
#define INFO_MASTER 0x001
#define INFO_COPY   0x002
#define INFO_CACHE  0x003
    char                ent_pepsycopy;
    time_t              ent_age;
    struct entrystruct  *ent_next;
} entrystruct, EntryInfo;
```

The parameters are used as follows:

`ent_dn`: The DN of the entry.

`ent_attr`: The requested attributes of the requested entry.

`ent_iscopy`: This is either `INFO_MASTER` or `INFO_COPY` indicating whether master data or copied data was used to generate the results. `INFO_CACHE` is not applicable to a DUA.

`ent_pepsycopy`: Unused.

`ent_age`: This field is the age of the entry, used to expire cached information.

`ent_next`: There may be many results, this is used to link them.

17.8 Compare

The compare operation is used to compare an asserted attribute with an attribute in the directory.

```
ds_compare (arg, error, result)
    struct ds_compare_arg       * arg;
    struct ds_compare_result    * result;
    struct DSError              * error;
```

You will need to include *quipu/compare.h* to use this procedure.
The argument is as follows:

```
struct ds_compare_arg {
    CommonArgs cma_common;
    DN         cma_object;
    AVA        cma_purported;
};
```

`cma_common`: The common arguments as described in Section 17.2.1.

`cma_object`: The DN of the object you want to compare an attribute against.

`cma_purported`: The attribute you want to compare. This structure is defined in Section 17.8.1.

If successful, the result structure is as follows:

```
struct ds_compare_result {
    CommonResults cmr_common;
    DN            cmr_object;
    char          cmr_matched;
    char          cmr_iscopy;
    char          cmr_pepsycopy;
    time_t        cmr_age;
};
```

`cmr_common`: The common results as described in Section 17.2.2.

`cmr_object`: The DN of the entry the attribute was compared against. This may not be the same as the DN supplied in the argument (e.g., if an alias was dereferenced).

`cmr_matched`: If `TRUE` then the attributes were the same, otherwise they were different.

`cmr_iscopy`: If `TRUE` then the compare took place against a copy of the attribute.

`cmr_pepsycopy`: Unused.

`cmr_age`: Unused.

17.8.1 Attribute Value Assertion

An "Attribute Value Assertion" is used by some of the operations to specify an attribute type/value pair, the structure used in this case is shown below.

```
typedef struct {
    AttributeType  ava_type;
    AttributeValue ava_value;
}AVA;
```

17.9 List

The list operation is shown below. This returns a list of subordinates of the specified entry.

```
ds_list (arg, error, result)
    struct ds_list_arg          * arg;
    struct ds_list_result       * result;
    struct DSError              * error;
```

You will need to include *quipu/list.h* to use this procedure.

```
struct ds_list_arg {
    CommonArgs lsa_common;
    DN         lsa_object;
};
```

`lsa_common`: The common arguments as described in Section 17.2.1.

`lsa_object`: The DN of the object you want to list the children of.

The results of a successful list form the following structure:

```
struct ds_list_result {
    CommonResults        lsr_common;
    DN                   lsr_object;
    time_t               lsr_age;
    struct subordinate * lsr_subordinates;
    POQ                  lsr_poq;
    struct ds_list_result * lsr_next;
};
```

`lsr_common`: The common results as described in Section 17.2.2.

`lsr_object`: The DN of the entry whose children were listed. This may not be the same as the DN supplied in the argument (e.g., if an alias was dereferenced).

`lsr_age`: Not currently used.

`lsr_subordinates`: This structure contains the RDNs that were found below the base object. The structure is as follows:

```
struct subordinate {
    RDN                 sub_rdn;
    char                sub_aliasentry;
    char                sub_copy;
    struct subordinate * sub_next;
};
```

`sub_rdn`: The RDN of this child.

`sub_aliasentry`: If `TRUE` then it is an alias entry.

`sub_copy`: If `TRUE` then the list took place against a copy of the object.

`sub_next`: A pointer to the next element in the list.

`lsr_poq`: A partial outcome qualifier. The structure is as follows:

```
#define LSR_NOLIMITPROBLEM      -1
#define LSR_TIMELIMITEXCEEDED   0
#define LSR_SIZELIMITEXCEEDED   1
#define LSR_ADMINSIZEEXCEEDED   2
typedef struct {
   int             poq_limitproblem;
   ContinuationRef poq_cref;
   char            poq_no_ext;
} POQ;
```

`poq_limitproblem`: Used to indicates which type of limit has been reached.

`poq_cref`: A list of continuation references. Continuation References are described in Section 17.3.

`poq_no_ext`: Used by *pepsy*.

`lsr_next`: A pointer to the next element if the list is uncorrelated. QUIPU will always return correlated results..

17.10 Search

The search operation performs a key directory functionality. This is done on the basis of filters as described in Section 17.10.1 below. The operation is defined as:

```
ds_search (arg, error, result)
    struct ds_search_arg        * arg;
    struct ds_search_result     * result;
    struct DSError              * error;
```

You will need to include *quipu/ds_search.h* to use this procedure.

```
struct ds_search_arg {
    CommonArgs          sra_common;
    DN                  sra_baseobject;
    int                 sra_subset;
#define SRA_BASEOBJECT          0
#define SRA_ONELEVEL            1
#define SRA_WHOLESUBTREE        2
    Filter              sra_filter;
    char                sra_searchaliases;
    EntryInfoSelection sra_eis;
};
```

`sra_common`: The common arguments as described in Section 17.2.1.

`sra_baseobject`: The DN of the object you want to start the search from.

`sra_subset`: This specifies which part of the DIT to search.

`sra_filter`: The filter to apply to the searched entries to see if the entry is required. Filters are discussed in Section 17.10.1

`sra_searchaliases`: If `TRUE`, aliases encountered below the search base object are dereferenced, and the dereferenced object searched. Note: this is different from the service control `dontderferencealiases` which is used to control dereferencing of entities above the base object.

`sra_eis`: The "Entry Information Selection", which defines which attributes you want to be returned (if the filter is matched), this is defined in Section 17.7.1.

The results form the following structure:

```
struct ds_search_result {
    char  srr_correlated;
    union {
        struct ds_search_unit   * srr_unit;
        struct ds_search_result * srr_parts;
    }     srr_un;
    struct ds_search_result * srr_next;
};
```

`srr_correlated`: If `TRUE` the the results are said to be correlated; that is, there will only be one element in the list of search results. A DSA may return uncorrelated result, in which case the routine

```
correlate_search_results(sr_res)
struct ds_search_result * sr_res;
```

can be called, which will correlate the results.

Note: QUIPU DSAs will always return correlated results.

`srr_un`: If the results are uncorrelated then the union will contain a nested `ds_search_result` structure, which contains a list of the uncorrelated results.

If the results are correlated, then this union will contain a pointer to the correlated results. These form the structure

```
struct ds_search_unit {
    CommonResults   srr_common;
    DN              srr_object;
    EntryInfo     * srr_entries;
    POQ             srr_poq;
};
#define CSR_common       srr_un.srr_unit->srr_common
#define CSR_object       srr_un.srr_unit->srr_object
```

```
#define CSR_entries      srr_un.srr_unit->srr_entries
#define CSR_limitproblem srr_un.srr_unit->\
                             srr_poq.poq_limitproblem
#define CSR_cr           srr_un.srr_unit->\
                             srr_poq.poq_cref
```

Note: the `#define`s to access the elements of this structure. The fields are used as follows:

`srr_common`: The common results as described in Section 17.2.2.

`srr_object`: The DN of the entry used as the base object of the search. This may not be the same as the DN supplied in the argument (e.g., if an alias was dereferenced).

`srr_entries`: A pointer to the "Entry Information", which will contain the attributes you requested (if present in the entry). The `EntryInfo` structure is defined in Section 17.7.2.

`srr_poq`: A partial outcome qualifier. This is the same as for the list operation described in the previous section.

`srr_next`: This is a pointer to the next result in a list of uncorrelated results.

17.10.1 Filters

A filter in its simplest sense is a single `filter_item`. This is used to perform one of six basic tests on one attribute of an entry. If the "match" is good, then the entry is returned as part of the search result structure. The six basic types of `filter_item` are represented by the structure below:

```
struct filter_item {
    int         fi_type;
#define FILTERITEM_EQUALITY       1
#define FILTERITEM_SUBSTRINGS     2
#define FILTERITEM_GREATEROREQUAL 3
#define FILTERITEM_LESSOREQUAL    4
#define FILTERITEM_PRESENT        5
#define FILTERITEM_APPROX         6
    union {
```

```
        AttributeType     fi_un_type;
        AVA               fi_un_ava;
        Filter_Substrings fi_un_substrings;
    }           fi_un;
    IFP         fi_ifp;
};
```

`fi_type`: Defines the type of `filter_item` being represented.

`fi_un.fi_un_type`: `FILTERITEM_PRESENT` matches. Just supply an AttributeType, an entry matches if this attribute exists within the entry.

`fi_un.fi_un_ava`: `FILTERITEM_EQUALITY`,
`FILTERITEM_GREATEROREQUAL`,
`FILTERITEM_LESSOREQUAL`,
and `FILTERITEM_APPROX` searches all take an Attribute Value Assertion (AVA) structure, and return the entries for which the attribute in the entry matches the asserted value. The AVA structure is defined in Section 17.8.1.

`fi_un.un_substrings`: `FILTERITEM_SUBSTRING` matches. Use the substring structure defined below, to specify which substrings to look for in the appropriate attribute.

```
typedef struct {
            AttributeType fi_sub_type;
            AV_Sequence   fi_sub_initial;
            AV_Sequence   fi_sub_any;
            AV_Sequence   fi_sub_final;
            char         *fi_sub_match;
        } Filter_Substrings;
```

The `AV_Sequence` structure is used to represent a string, and should be treated as essentially a linked list of `char *` parameters.

`fi_sub_type`: The attribute that you want to perform a substring search on.

`fi_sub_initial`: This contains a single attribute value, that must appear at the start of the string.

`fi_sub_any`: A set of values, which must appear in order in the middle of the string.

`fi_sun_final`: This contains a single attribute value, that must appear at the end of the string.

`fi_sub_match`: For DSA use only.

`fi_ifp`: For DSA use only.

The single filter items can now be linked into a `filter` structure to build more complex search definitions:

```
typedef struct s_filter {
    char          flt_type;
#define FILTER_ITEM 1
#define FILTER_AND  2
#define FILTER_OR   3
#define FILTER_NOT  4
   struct filter * flt_next;
   union {
        struct filter_item   flt_un_item;
        struct filter      * flt_un_filter;
   }               flt_un;
}s_filter, *Filter;
```

`flt_type`: This defines whether the filter is a single item, or a more complex filter made up of one or more components.

`flt_un.flt_un_item`: If the filter represents a `filter_item`, then the item is placed here.

`flt_un.flt_un_filter`: `AND`, `OR` and `NOT` filters apply to a linked list of "children" filters. This element is a pointer to the head of that list.

`flt_next`: If the parent filter is an `AND`, `OR`. or `NOT` filter, then the component filters are linked using this field. *Note*: that `NOT` filters should contain a list of one child only.