```
struct FADUidentity {
    int     fa_type;
#define FA_FIRSTLAST    0       /* first-last */
#define FA_RELATIVE     1       /* relative */
#define FA_BEGINEND     2       /* begin-end */
#define FA_SINGLE       3       /* single-name */
#define FA_NAMELIST     4       /* name-list */
#define FA_FADUNUMBER   5       /* fadu-number */

    union {
        int     fa_un_firstlast;
#define FA_FIRST        0
#define FA_LAST         1

        int     fa_un_relative;
#define FA_PREVIOUS     0
#define FA_CURRENT      1
#define FA_NEXT         2

        int     fa_un_beginend;
#define FA_BEGIN        0
#define FA_END          1

        char   *fa_un_singlename;

#define NANAME  5
        struct {
            char   *fa_un_names[NANAME];
            int     fa_un_nname;
        }       fa_un_list;

        int     fa_un_fadunumber;
    }   fa_un;
#define fa_firstlast    fa_un.fa_un_firstlast
#define fa_relative     fa_un.fa_un_relative
#define fa_beginend     fa_un.fa_un_beginend
#define fa_singlename   fa_un.fa_un_singlename
#define fa_names        fa_un.fa_un_list.fa_un_names
#define fa_nname        fa_un.fa_un_list.fa_un_nname
```

```
#define fa_fadunumber   fa_un.fa_un_fadunumber
};
```

As shown, this structure is really a discriminated union (a structure with a tag element followed by a union). The elements are:

`fa_type`: indicates what type of FADU identity is represented; Depending on this type, one of the following elements is valid:

`fa_firstlast`: the location, identified as first or last;

`fa_relative`: the location, identified as previous, current, or next;

`fa_beginend`: the location, identified as begin or end;

`fa_singlename`: the location, as identified by name;

`fa_names/fa_nname`: the location, as identified by a list of names (and the length of that list, which may not exceed the manifest constant `NANAME`); or,

`fa_fadunumber`: the location, as identified by number.

A limitation of this implementation is that a fixed number of FADU names may be supported, as denoted by the manifest constant `NANAME`.

The `FUFREE` macro can be used to free the storage associated with an `FADUidentity` structure (without freeing the structure itself). It behaves as if it was defined as:

```
void    FUFREE (fu)
struct FADUidentity *fu;
```

2.4 Association Establishment

The *libftam*(3n) library distinguishes between the entity which started an association, the *initiator*, and the entity which was subsequently bound to the association, the *responder*. We sometimes term these two entities the *client* and the *server*, respectively.

Section 2.2.1 of *Volume One* describes the use of addressing for the application layer. In particular, Figure 2.1 on page 21 of *Volume One* presents an example of how one constructs the address for the File Transfer, Access and Management (FTAM) service on host `RemoteHost`.

2.4.1 Responder

The *tsapd*(8c) daemon, upon accepting a connection from an initiating host, consults the ISO services database to determine which program on the local system implements the desired application context.

Once the program has been ascertained, the daemon runs the program with any arguments listed in the database. In addition, it appends some *magic arguments* to the argument vector. Hence, the very first action performed by the responder is to re-capture the FTAM state contained in the magic arguments. This is done by calling the routine `FInit`, which on a successful return, is equivalent to an F-INITIALIZE.INDICATION event.

```
int     FInit (vecp, vec, fts, tracing, fti)
int     vecp;
char  **vec;
struct FTAMstart *fts;
int   (*tracing) ();
struct FTAMindication *fti;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

`fts`: a pointer to an `FTAMstart` structure, which is updated only if the call succeeds;

`tracing`: the address of a tracing routine to be invoked when an FTAM event occurs (consult Section 2.5.3); and,

`fti`: a pointer to an `FTAMindication` structure, which is updated only if the call fails.

If `FInit` is successful, it returns information in the `fts` parameter, which is a pointer to an `FTAMstart` structure.

```
struct FTAMstart {
    int     fts_sd;

    AEInfo  fts_calledtitle;
```

```
    AEInfo  fts_callingtitle;

    struct PSAPaddr fts_calledaddr;
    struct PSAPaddr fts_callingaddr;

    OID     fts_context;

    int     fts_manage;

    int     fts_class;

    int     fts_units;
    int     fts_attrs;

    PE      fts_sharedASE;

    int     fts_fqos;

    struct FTAMcontentlist fts_contents;

    char   *fts_initiator;
    char   *fts_account;
    char   *fts_password;
    int     fts_passlen;

    int     fts_ssdusize;

    struct QOStype fts_qos;
};
```

The elements of this structure are:

`fts_sd`: the association-descriptor to be used to reference this association;

`fts_calledtitle`: information on the called application-entity, if any (consult Section 2.2.1 of *Volume One*);

`fts_callingtitle`: information on the the calling application-entity, if any;

`fts_calledaddr`: the called presentation address (consult Section 2.2 of *Volume Two*);

`fts_callingaddr`: the calling presentation address;

`fts_context`: the application context name;

`fts_manage`: a flag indicating if presentation context management is available for use (zero if unavailable);

`fts_class`: the inclusive-or of the service classes offered;

`fts_units`: the functional units;

`fts_attrs`: the attribute groups;

`fts_sharedASE`: shared ASE information;

`fts_fqos`: the FTAM Quality-of-Service;

`fts_contents`: the contents type list;

`fts_initiator`: the user-identity of the initiator, if any;

`fts_account`: the account, if any;

`fts_password/fts_passlen`: the password (and its length), if any;

`fts_ssdusize`: the largest atomic SSDU size that can be used on the underlying connection; and,

`qos`: the quality of service on the connection (see Section 4.6.2 in *Volume Two*).

Note that the data contained in the structure was allocated via *malloc*(3), and should be released by using the `FTSFREE` macro when no longer referenced. The `FTSFREE` macro, behaves as if it was defined as:

```
void    FTSFREE (fts)
struct FTAMstart *fts;
```

The macro frees only the data allocated by `FInit`, and not the `FTAMstart` structure itself. Further, `FTSFREE` should be called only if the call to the `FInit` routine returned `OK`.

If the call to `FInit` is not successful, then an F-P-ABORT.INDICATION event is simulated, and the relevant information is returned in an encoded `FTAMindication` structure.

```
struct FTAMindication {
    int     fti_type;
#define FTI_FINISH      0x00
#define FTI_ABORT       0x01
#define FTI_MANAGEMENT  0x02
#define FTI_BULKBEGIN   0x03
#define FTI_BULKEND     0x04
#define FTI_ACCESS      0x05
#define FTI_READWRITE   0x06
#define FTI_DATA        0x07
#define FTI_DATAEND     0x08
#define FTI_CANCEL      0x09
#define FTI_TRANSEND    0x10

    union {
        struct FTAMfinish   fti_un_finish;
        struct FTAMabort    fti_un_abort;
        struct FTAMgroup    fti_un_group;
        struct FTAMaccess   fti_un_access;
        struct FTAMreadwrite fti_un_readwrite;
        struct PSAPdata     fti_un_data;
        struct FTAMdataend  fti_un_dataend;
        struct FTAMcancel   fti_un_cancel;
        struct FTAMtransend fti_un_transend;
    }   fti_un;
#define fti_finish      fti_un.fti_un_finish
#define fti_abort       fti_un.fti_un_abort
#define fti_group       fti_un.fti_un_group
#define fti_access      fti_un.fti_un_access
#define fti_readwrite   fti_un.fti_un_readwrite
#define fti_data        fti_un.fti_un_data
#define fti_dataend     fti_un.fti_un_dataend
```

```
#define fti_cancel      fti_un.fti_un_cancel
#define fti_transend    fti_un.fti_un_transend
};
```

As shown, this structure is really a discriminated union (a structure with a tag element followed by a union). Hence, on a failure return, one first coerces a pointer to the `FTAMabort` structure contained therein, and then consults the elements of that structure.

```
struct FTAMabort {
    int     fta_peer;

    int     fta_action;

#define NFDIAG  5
    int     fta_ndiag;
    struct FTAMdiagnostic fta_diags[NFDIAG];
};
```

The elements of an `FTAMabort` structure are:

`fta_peer`: if set, indicates that a user-initiated abort occurred (an F-U-ABORT.INDICATION event); if not set, indicates that a provider-initiated abort occurred (an F-P-ABORT.INDICATION event);

`fta_action`: an action result; and,

`fta_diags/fta_ndiag`: any diagnostic information (and the number of diagnostics present).

For each diagnostic present (typically only one) in the `FTAMabort` structure, information regarding the failure is present. In particular, the error code can be found in the the `ftd_identifier` element of each diagnostic.

After examining the information returned by `FInit` on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the association. For either response, the responder should use the `FInitializeResponse` routine (which corresponds to the F-INITIALIZE.RESPONSE action).

```
int     FInitializeResponse (sd, state, action, context,
        respondtitle, respondaddr, manage, class, units,
        attrs, sharedASE, fqos, contents, diag, ndiag, fti)
int     sd;
int     state,
        action,
        manage,
        class,
        units,
        attrs,
        fqos;
OID     context;
AEI     respondtitle;
struct PSAPaddr *respondaddr;
PE      sharedASE;
struct FTAMcontentlist *contents;
struct FTAMdiagnostic diag[];
int     ndiag;
struct FTAMindication *fti;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`state:` a state result;

`action:` an action result;

`context:` the application context name (use the manifest constant `NULLOID` if the proposed name should be used);

`manage:` the negotiated use of presentation context management;

`respondtitle:` information on the responding application-entity, if any;

`respondaddr:` the responding presentation address, if any;

`manage:` the negotiated service class;

`units:` the negotiated functional units (specify both mandatory and optional functional units for the service class);

`attrs`: the negotiated attribute groups;

`sharedASE`: shared ASE information;

`fqos`: the negotiated FTAM Quality-of-Service;

`contents`: the negotiated contents type list;

`diag/ndiag`: a list of diagnostics (and the number of diagnostics in the list); and,

`fti`: a pointer to an `FTAMindication` structure, which is updated only if the call fails.

If the call to `FInitializeResponse` is successful, then association establishment has now been completed. Otherwise, if the call failed and the reason is "fatal", then the association is lost.

2.4.2 Initiator

A program wishing to associate itself with a filestore calls the routine `FInitializeRequest`, which corresponds to the user taking the F-INITIALIZE.REQUEST action.

```
int     FInitializeRequest (context, callingtitle, calledtitle,
        callingaddr, calledaddr, manage, class, units, attrs,
        sharedASE, fqos, contents, initiator, account,
        password, passlen, qos, tracing, ftc, fti)
OID     context;
AEI     calledtitle,
        callingtitle;
struct PSAPaddr *calledaddr,
                *callingaddr;
int     manage,
        class,
        units,
        attrs,
        fqos,
        passlen;
PE      sharedASE;
struct FTAMcontentlist *contents;
```

```
char   *initiator,
       *account,
       *password;
struct QOStype *qos;
IFP     tracing;
struct FTAMconnect *ftc;
struct FTAMindication *fti;
```

The parameters to this procedure are:

`context`: the application context name (use the manifest constant `NULLOID` to invoke the default file transfer context);

`callingtitle`: information on the the calling application-entity, if any (consult Section 2.2.1 of *Volume One*);

`calledtitle`: information on the called application-entity, if any;

`callingaddr`: the calling presentation address (consult Section 2.2 of *Volume Two*);

`calledaddr`: the called presentation address;

`manage`: a flag indicating if presentation context management is available for use (zero if unavailable);

`class`: the proposed service classes;

`units`: the proposed functional units (specify both mandatory and optional functional units for the indicated service class);

`attrs`: the proposed attribute groups;

`sharedASE`: shared ASE information;

`fqos`: the proposed FTAM Quality-of-Service;

`contents`: the contents type list;

`initiator`: the user-identity of the initiator, if any;

`account`: the account, if any