```
AEI     _str2aei (designator, qualifier, context, interactive)
char   *designator,
       *qualifier,
       *context;
int     interactive;
```

The parameters to this procedure are:

`designator:` input from the user (e.g., "`hubris, cs, ucl, gb`" or "`hubris`");

`qualifier:` the service qualifier (e.g., `filestore`);

`context:` the `supportedApplicationContext` for the desired service (e.g., `iso ftam`); and;

`interactive:` non-zero if the user's terminal can be queried for additional information.

The routine `_str2aei` will first attempt to resolution using the "user-friendly nameservice" with the `designator`, `context`, and `interactive` parameters. If this fails, the "stub directory service" will be used, with the `designator` and `qualifier` parameters.

For backwards-compatiblity, a macro, `str2aei` is provided which is equivalent to:

```
_str2aei (designator, qualifier, NULLCP, 0)
```

For future-compability, you should start using the `str2aeinfo` routine, as both `_str2aei` and its macro-ized counterpart will soon be deprecated:

```
AEI     str2aeinfo (string, context, interactive, userdn, passwd)
char   *string,
       *context,
       *userdn,
       *passwd;
int     interactive;
```

The parameters to this procedure are:

`string:` input from the user (e.g., "`hubris, cs, ucl, gb`" or "`hubris`");

`context`: the `supportedApplicationContext` for the desired service (e.g., `iso ftam`);

`interactive`: non-zero if the user's terminal can be queried for additional information;

`userdn`: the Distinguished Name (DN) when using the Directory service; and;

`passwd`: authentication information for the Directory service.

This routine attempts resolution solely via the "user-friendly nameservice".

The routine `aei2addr` takes application-entity information, and returns the appropriate presentation address.

```
struct PSAPaddr *aei2addr (aei)
AEI     aei;
```

Address Encodings

It may be useful to encode a presentation address for viewing. Although a consensus for a standard way of doing this has not yet been reached, the routines `paddr2str` and `str2paddr` may be used in the interim. These implement the encodings defined in [Kil89]. The BNF syntax for this encoding is shown in Figure 2.2 on page 23.

```
char   *paddr2str (pa, na)
struct PSAPaddr *pa;
struct NSAPaddr *na;
```

The parameters to this procedure are:

`pa`: the presentation address; and,

`na`: a network address to use instead of the network addresses in the `pa` parameter (use the manifest constant `NULLNA` otherwise).

If `paddr2str` fails, it returns the manifest constant `NULLCP`.

The routine `str2paddr` takes an ascii string encoding and returns a presentation address.

```
#include <isode/acsap.h>

...

register struct PSAPaddr *pa;
register AEI aei;

...

char *qualifier = "filestore";                                          10
char *context = "iso ftam";

if ((aei = _str2aei ("hubris, cs, ucl, gb", qualifier,
                        context, isatty (fileno (stdin)),
                        NULLCP, NULLCP))
          == NULLAEI)
    error ("unable to resolve service:  %s", PY_pepy);
if ((pa = aei2addr (aei)) == NULLPA)
    error ("address translation failed");
                                                                        20
...
```

Figure 2.1: Constructing the address for the FTAM entity

```
struct PSAPaddr *str2paddr (str)
char   *str;
```

The parameter to this procedure is:

`str:` the ascii string.

If `str2paddr` fails, it returns the manifest constant `NULLPA`.

`paddr2str` is really a macro which calls the routine `_paddr2str`:

```
char  *_paddr2str (pa, na, compact)
struct PSAPaddr *pa;
struct NSAPaddr *na;
int    compact;
```

The parameters to this procedure are:

`pa:` the presentation address;

`na:` a network address to use instead of the network addresses in the `pa` parameter (use the manifest constant `NULLNA` otherwise); and,

`compact:` indicates the style of encoding.

If `compact` is greater than zero, then the encoding reflects a normalized network address. This is useful for passing to arbitrary processes in the network. If `compact` is less than zero, then the encoding reflects the BNF in Figure 2.2, without any macro substitutions. If `compact` is equal to zero (which is what `paddr2str` uses), then the encoding reflects the above BNF with macro substitutions.

The routine `pa2str` takes a presentation address and returns a null-terminated ascii string suitable for viewing:

```
char   *pa2str (pa)
struct PSAPaddr *pa;
```

The parameter to this procedure is:

`pa` : the presentation address to be printed.

```
<presentation-address> ::=
                [[[ <psel> "/" ] <ssel> "/" ] <tsel> "/" ]
                <network-address-list>

<network-address-list> ::=
                <network-address> "|" <network-address-list>
            | <network-address>

<psel> ::=      <selector>
<ssel> ::=      <selector>                                          10
<tsel> ::=      <selector>

<selector> ::=
                '"' <otherstring> '"'           -- IA5
                                                -- (for other chars use hex)
            | "#" <digitstring>                 -- US GOSIP
            | "'" <hexstring> "'H"              -- hex
            | ""                                -- empty but present

<network-address> ::=                                              20
                "NS" "+" <hexstring>            -- concrete binary
                                                -- this is the compact
                                                --   encoding
            | <afi> "+" <idi> [ "+" <dsp> ]


<afi>           ::= "X121" | "DCC" | "TELEX" | "PSTN" | "ISDN"
                  | "ICD" | "LOCAL"

<idi>           ::= <digitstring>                                  30

<dsp>  ::=      "d" <digitstring>               -- abstract decimal
            | "x" <hexstring>                   -- abstract binary
            | "l" <otherstring>                 -- IA5: local form only
            | "RFC-1006" "+" <prefix> "+" <ip>
                  [ "+" <port> [ "+" <tset> ]]
            | "X.25(80)" "+" <prefix> "+" <dte>
                  [ "+" <cudf-or-pid> "+" <hexstring> ]
            | "ECMA-117-Binary" "+" <hexstring> "+" <hexstring>
                  "+" <hexstring>                                  40
            | "ECMA-117-Decimal" "+" <digitstring> "+" <digitstring>
                  "+" <digitstring>
```

Figure 2.2: BNF Syntax for paddr2str/str2paddr

```
<prefix>          ::= <digit> <digit>

<ip>              ::= <otherstring>
                          -- dotted decimal form (e.g., 10.0.0.6)
                          -- or domain (e.g., twg.com)
<port>            ::= <digitstring>
<tset>            ::= <digitstring>

<dte>             ::= <digitstring>
<cudf-or-pid>     ::= "CUDF" | "PID"                                    10

<digitstring>     ::= <digit>    <digitstring> | <digit>
<digit>           ::= [0-9]

<otherstring>     ::= <other>    <otherstring> | <other>
<other>           ::= [0-9a-zA-Z+-.]

<hexstring>       ::= <hexdigit> <hexstring>   | <hexdigit>
<hexdigit>        ::= [0-9a-f]
```

Figure 2.2: BNF Syntax for paddr2str/str2paddr (continued)

Server Initialization

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 ACSE state contained in the magic arguments. This is done by calling the routine `AcInit`, which on a successful return, is equivalent to a A-ASSOCIATE.INDICATION event from the association control service provider.

```
int     AcInit (vecp, vec, acs, aci)
int     vecp;
char  **vec;
struct AcSAPstart *acs;
struct AcSAPindication *aci;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

`acs`: a pointer to an `AcSAPstart` structure, which is updated only if the call succeeds; and,

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

If `AcInit` is successful, it returns information in the `acs` parameter, which is a pointer to an `AcSAPstart` structure.

```
struct AcSAPstart {
    int     acs_sd;

    OID     acs_context;

    AEInfo  acs_callingtitle;
    AEInfo  acs_calledtitle;
```

```
    struct PSAPstart acs_start;

    int     acs_ninfo;
    PE      acs_info[NACDATA];
};
```

The elements of this structure are:

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

`acs_context:` the application context name;

`acs_callingtitle:` information on the calling application-entity (if any);

`acs_calledtitle:` information on the called application-entity (if any);

`acs_start:` an `PSAPstart` structure (consult page 20 of *Volume Two*); and,

`acs_info/acs_ninfo:` any initial data (and the length of that data).

Note that the proposed contexts list in the `acs_start` element will contain a context for the ACSE. The routine `AcFindPCI` can be used to determine the PCI being used by the ACSE:

```
int     AcFindPCI (sd, pci, aci)
int     sd;
int    *pci;
struct AcSAPindication *aci;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`pci:` a pointer to an integer location, which is updated only if the call succeeds; and,

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

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

```
void    ACSFREE (acs)
struct AcSAPstart *acs;
```

The macro frees only the data allocated by AcInit, and not the AcSAPstart structure itself. Further, ACSFREE should be called only if the call to the AcInit routine returned OK.

If the call to AcInit is not successful, then a A-P-ABORT.INDICATION event is simulated, and the relevant information is returned in an encoded AcSAPindication structure.

```
struct AcSAPindication {
    int     aci_type;
#define ACI_FINISH      0x00
#define ACI_ABORT       0x01

    union {
        struct AcSAPfinish aci_un_finish;
        struct AcSAPabort aci_un_abort;
    }   aci_un;
#define aci_finish      aci_un.aci_un_finish
#define aci_abort       aci_un.aci_un_abort
};
```

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 AcSAPabort structure contained therein, and then consults the elements of that structure.

```
struct AcSAPabort {
    int     aca_source;

    int     aca_reason;

    int     aca_ninfo;
    PE      aca_info[NACDATA];
```

```
#define ACA_SIZE        512
    int     aca_cc;
    char    aca_data[ACA_SIZE];
};
```

The elements of an `AcSAPabort` structure are:

`aca_source`: the source of the abort, one of:

Value	Source
`ACA_USER`	service-user (peer)
`ACA_PROVIDER`	service-provider
`ACA_LOCAL`	local ACPM

`aca_reason`: the reason for the provider-initiated abort (meaningful only if `aca_source` is not `ACA_REQUESTOR`), consult Table 2.1;

`aca_info/aca_ninfo`: any abort data (and the length of that data) from the peer (if `aca_source` is `ACA_USER`); and,

`aca_data/aca_cc`: a diagnostic string from the provider.

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

```
void    ACAFREE (aca)
struct AcSAPabort *aca;
```

The macro frees only the data contained in the structure, and not the `AcSAPabort` structure itself.

After examining the information returned by `AcInit` 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 `AcAssocResponse` routine (which corresponds to the A-ASSOCIATE.RESPONSE action).

```
int     AcAssocResponse (sd, status, reason, context,
        respondtitle, respondaddr, ctxlist, defctxresult,
        prequirements, srequirements, isn, settings, ref,
```