Provider-Initiated Aborts (fatal)	
ACS_ADDRESS	Address unknown
ACS_REFUSED	Connect request refused on this network connection
ACS_CONGEST	Local limit exceeded
ACS_PRESENTATION	Presentation disconnect
ACS_PROTOCOL	Protocol error
ACS_RESPONDING	Rejected by responding ACPM
ACS_ABORTED	Peer aborted association
User-Initiated Rejections (fatal)	
ACS_PERMANENT	Permanent
ACS_TRANSIENT	Transient
User-Initiated Rejections (non-fatal)	
ACS_REJECT	Release rejected
Interface Errors (non-fatal)	
ACS_PARAMETER	Invalid parameter
ACS_OPERATION	Invalid operation

Table 2.1: AcSAP Failure Codes

```
        data, ndata, aci)
int     sd;
int     status,
        reason;
OID     context;
AEI     respondtitle;
struct PSAPaddr *respondaddr;
int     prequirements,
        srequirements,
        settings,
        ndata;
long    isn;
struct PSAPctxlist *ctxlist;
int     defctxresult;
struct SSAPref *ref;
PE     *data;
struct AcSAPindication *aci;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`status`: the acceptance indicator (either `ACS_ACCEPT` if the association is to be accepted, or one of the user-initiated rejection codes listed in Table 2.1 on page 29);

`reason`: the diagnostic (either `ACS_USER_NULL` if the association is to be accepted, or one of the user-initiated diagnostic codes listed in Table 2.2 on page 31);

`context`: the application context to be used over the association (defaults to the context proposed);

`respondtitle`: information on the responding application-entity (optional);

`data/ndata`: an array of user data (and the number of elements in that array, consult the warning on page 49); and,

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

Provider-Initiated Diagnostics	
ACS_PROV_NULL	Null
ACS_PROV_NOREASON	No reason given
ACS_VERSION	No common acse version
User-Initiated Diagnostics	
ACS_USER_NULL	Null
ACS_USER_NOREASON	No reason given
ACS_CONTEXT	Application context name not supporte
ACS_CALLING_AP_TITLE	Calling AP title not recognized
ACS_CALLING_AP_ID	Calling AP invocation-ID not recognized
ACS_CALLING_AE_QUAL	Calling AE qualifier not recognized
ACS_CALLING_AE_ID	Calling AE invocation-ID not recognized
ACS_CALLED_AP_TITLE	Called AP title not recognized
ACS_CALLED_AP_ID	Called AP invocation-ID not recognized
ACS_CALLED_AE_QUAL	Called AE qualifier not recognized
ACS_CALLED_AE_ID	Called AE invocation-ID not recognized

Table 2.2: AcSAP Diagnostic Codes

The remaining parameters are for the presentation service, consult the description of the `PConnResponse` routine on page 27 of *Volume Two*.

If the call to `AcAssocResponse` is successful, and the `status` parameter is set to `ACS_ACCEPT`, then association establishment has now been completed. If the call is successful, but the `status` parameter is not `ACS_ACCEPT`, then the association has been rejected and the responder may exit. Otherwise, if the call failed and the reason is "fatal", then the association is lost.

Client Initialization

A program wishing to associate itself with another application-level user calls the `AcAssocRequest` routine, which corresponds to the user taking the A-ASSOCIATE.REQUEST action.

```
int     AcAssocRequest (context, callingtitle, calledtitle,
        callingaddr, calledaddr, ctxlist, defctxname,
        prequirements, srequirements, isn, settings, ref,
        data, ndata, qos, acc, aci)
OID     context;
AEI     callingtitle,
        calledtitle;
struct PSAPaddr *callingaddr,
                *calledaddr;
int     prequirements,
        srequirements,
        settings,
        ndata;
long    isn;
struct PSAPctxlist *ctxlist;
OID     defctxname;
struct SSAPref *ref;
PE    *data;
struct QOStype *qos;
struct AcSAPconnect *acc;
struct AcSAPindication *aci;
```

The parameters to this procedure are:

`context`: the application context to be used over the association;

`callingtitle`: information on the calling application-entity (use the manifest constant `NULLAEI` if not specified);

`calledtitle`: information on the called application-entity (use the manifest constant `NULLAEI` if not specified);

`data/ndata`: an array of initial data (and the number of elements in that array, consult the warning on page 49);

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

`acc`: a pointer to an `AcSAPconnect` 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.

The remaining parameters are for the presentation service, consult the description of the `PConnRequest` routine on page 29 of *Volume Two* for additional information. Note that the `ctxlist` parameter is mandatory: the association control service element will look for the ACSE PCI there. If not present, it will add the PCI to the list.

If the call to `AcAssocRequest` is successful (the A-ASSOCIATE.CONFIRMATION event occurs), it returns information in the `acc` parameter which is a pointer to an `AcSAPconnect` structure.

```
struct AcSAPconnect {
    int     acc_sd;

    int     acc_result;
    int     acc_diagnostic;

    OID     acc_context;

    AEInfo  acc_respondtitle;

    struct PSAPconnect acc_connect;

    int     acc_ninfo;
    PE      acc_info[NACDATA];
};
```

The elements of an `AcSAPconnect` structure are:

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

`acc_result/acc_diagnostic`: the association response and diagnostic;

`acc_context`: the application context name;

`acc_respondtitle`: information on the responding application-entity (if any);

`acc_connect`: an `PSAPconnect` structure (consult page 30 of *Volume Two*); and,

`acc_info/acc_ninfo`: any initial data (and the length of that data).

Note that the negotiated contexts list in the `acc_connect` element will contain a context for the ACSE. To determine the PCI for the ACSE context, the routine `AcFindPCI` routine, described above can be used.

If the call to `AcAssocRequest` is successful, and the `acc_result` element is set to `ACC_ACCEPT`, then association establishment has completed. If the call is successful, but the `acc_result` element is not `ACC_ACCEPT`, then the association attempt has been rejected; consult Table 2.1 on page 29 and Table 2.2 on page 31 to determine the reason for the rejection (further information can be found in the `aci` parameter). Otherwise, if the call fails then the association is not established and the `AcSAPabort` structure of the `AsCAPindication` discriminated union has been updated.

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

```
void    ACCFREE (acc)
struct AcSAPconnect *acc;
```

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

Normally `AcAssocRequest` returns only after an association has succeeded or failed. This is termed a *synchronous* association initiation. If the user desires, an *asynchronous* association may be initiated. This routine is really a macro which calls the routine `AcAsynAssocRequest` with an argument indicating that a association should be attempted synchronously.

```
int     AcAsynAssocRequest (context, callingtitle, calledtitle,
        callingaddr, calledaddr, ctxlist, defctxname,
        prequirements, srequirements, isn, settings, ref,
        data, ndata, qos, acc, aci, async)
OID     context;
AEI     callingtitle,
        calledtitle;
```

```
struct PSAPaddr *callingaddr,
                *calledaddr;
int     prequirements,
        srequirements,
        settings,
        ndata,
        async;
long    isn;
struct PSAPctxlist *ctxlist;
OID     defctxname;
struct SSAPref *ref;
PE    *data;
struct QOStype *qos;
struct AcSAPconnect *acc;
struct AcSAPindication *aci;
```

The additional parameter to this procedure is:

`async`: whether the association should be initiated asynchronously.

If the `async` parameter is non-zero, then `AcAsynAssocRequest` returns one of four values: `NOTOK`, which indicates that the association request failed; `DONE`, which indicates that the association request succeeded; or, either of `CONNECTING_1` or `CONNECTING_2`, which indicates that the connection request is still in progress. In the first two cases, the usual procedures for handling return values from `AcAssocRequest` are employed (i.e., a `NOTOK` return from `AcAsynAssocRequest` is equivalent to a `NOTOK` return from `AcAssocRequest`, and, a `DONE` return from `AcAsynAssocRequest` is equivalent to a `OK` return from `AcAssocRequest`).

In the final case, when either `CONNECTING_1` or `CONNECTING_2` is returned, only the `acc_sd` element of the `acc` parameter has been updated; it reflects the association-descriptor to be used to reference this association. To determine when the association attempt has been completed, the routine `xselect` (consult Section 2.4) should be used after calling `PSelectMask`. In order to determine if the association attempt is successful, the routine `AcAsynRetryRequest` is called:

```
int     AcAsynRetryRequest (sd, acc, aci)
int     sd;
```

```
struct AcSAPconnect *acc;
struct AcSAPindication  *aci;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`acc:` a pointer to an `AcSAPconnect` 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.

Again, one of three values are returned: `NOTOK`, which indicates that the association request failed; `DONE`, which indicates that the association request succeeded; and, `CONNECTING_1` or `CONNECTING_2` which indicates that the connection request is still in progress.

Refer to Section 4.2.3 on page 112 in *Volume Two* for information on how to make efficient use of the asynchronous connection facility.

2.2.2 Association Release

The `AcRelRequest` routine is used to request the release of an association, and corresponds to a A-RELEASE.REQUEST action.

```
int     AcRelRequest (sd, reason, data, ndata, secs, acr, aci)
int     sd;
int     reason;
PE     *data;
int     ndata;
int     secs;
struct AcSAPrelease *acr;
struct AcSAPindication *aci;
```

The parameters to this procedure:

`sd:` the association-descriptor;

`reason:` the reason why the association should be released, one of:

Value	Reason
ACF_NORMAL	normal
ACF_URGENT	urgent
ACF_UNDEFINED	undefined

`data/ndata:` an array of final data (and the number of elements in that array, consult the warning on page 49);

`secs:` the maximum number of seconds to wait for a response (use the manifest constant `NOTOK` if no time-out is desired);

`acr:` a pointer to an `AcSAPrelease` 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 the call to `AcRelRequest` is successful, then this corresponds to a A-RELEASE.CONFIRMATION event, and it returns information in the `acr` parameter, which is a pointer to an `AcSAPrelease` structure.

```
struct AcSAPrelease {
    int     acr_affirmative;

    int     acr_reason;

    int     acr_ninfo;
    PE      acr_info[NACDATA];
};
```

The elements of this structure are:

`acr_affirmative:` the acceptance indicator;

`acr_reason:` the reason for the indicator, one of:

Value	Reason
ACR_NORMAL	normal
ACR_NOTFINISHED	not finished
ACR_UNDEFINED	undefined

`acr_info/acr_ninfo:` any final data (and the length of that data).

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

```
void    ACRFREE (acr)
struct AcSAPrelease *acr;
```

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

If the call to `AcRelRequest` is successful, and the `acr_affirmative` element is set, then the association has been released. If the call is successful, but the `acr_affirmative` element is not set (i.e., zero), then the request to release the association has been rejected by the remote user, and the association is still established. Otherwise the `AcSAPabort` element of the `AcSAPindication` parameter `aci` contains the reason for failure.

Note that if a non-negative value is given to the `secs` parameter and a response is not received within this number of seconds, then the value contained in the `aca_reason` element is `ACS_TIMER`. The user can then call the routine `AcRelRetryRequest` to continue waiting for a response:

```
int     AcRelRetryRequest (sd, secs, acr, aci)
int     sd;
int     secs;
struct AcSAPrelease  *acr;
struct AcSAPindication  *aci;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`secs:` the maximum number of seconds to wait for a response (use the manifest constant `NOTOK` if no time-out is desired);

`acr:` a pointer to a `AcSAPrelease` structure, which is updated only if the call succeeds; and,

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