If the call to AcRelRetryRequest is successful, and the acr_affirmative element is set, then the association has been closed. If the call is successful, but the acr_affirmative element is not set (i.e., zero), then the request to release the assocation has been rejected by the remote user, and the association is still open. Otherwise the AcSAPabort structure contained in the AcSAPindication parameter aci contains the reason for failure. As expected, the value ACS_TIMER indicates that no response was received within the time given by the secs parameter.

Upon receiving a A-RELEASE.INDICATION event, the user is required to generate a A-RELEASE.RESPONSE action using the AcRelResponse routine.

```
int     AcRelResponse (sd, status, reason, data, ndata, aci)
int     sd;
int     status,
        reason;
PE     *data;
int     ndata;
struct AcSAPindication *aci;
```

The parameters to this procedure:

sd: the association-descriptor;

status: the acceptance indicator (either ACS_ACCEPT if the association is to be released, or ACS_REJECT otherwise);

reason: the reason for the indicator, one of:

Value	Reason
ACR_NORMAL	normal
ACR_NOTFINISHED	not finished
ACR_UNDEFINED	undefined

data/ndata: an array of final 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.

If the call to `AcRelResponse` is successful, and if the `result` parameter is set to `ACS_ACCEPT`, then the association has been released. If the call is successful, but the `result` parameter is not `ACS_ACCEPT`, then the association remains established.

2.2.3 Association Abort

To unilaterally abort an association, the routine `AcUAbortRequest` routine is used which corresponds to the A-ABORT.REQUEST action.

```
int     AcUAbortRequest (sd, data, ndata, aci)
int     sd;
PE     *data;
int     ndata;
struct AcSAPindication *aci;
```

The parameters to this procedure:

`sd:` the association-descriptor;

`data/ndata:` an array of abort 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.

If the call to `AcUAbortRequest` is successful, then the association is immediately released, and any data queued for the association may be lost.

2.3 Association Events

Typically, the presentation service generates events which lead to the A-RELEASE.INDICATION, A-ABORT.INDICATION, and A-P-ABORT.INDICATION events being raised. For those interested in writing application service elements which interpret this information, this section describes the necessary mappings.

2.3.1 Release Indication

Upon receiving a P-RELEASE.INDICATION event, the routine `AcFINISHser` should be called to map this into a A-RELEASE.INDICATION event.

```
int     AcFINISHser (sd, pf, aci)
int     sd;
struct PSAPfinish *pf;
struct AcSAPindication *aci;
```

The parameters to this procedure:

`sd:` the association-descriptor;

`pa:` a pointer to an `PSAPfinish` structure, containing information on the presentation-level release; and,

`aci:` a pointer to an `AcSAPindication` structure.

If the call to `AcFINISHser` is successful, then the `aci_type` field of the `aci` parameter contains the value `ACI_FINISH`, and an `AcSAPfinish` structure is contained inside the `aci` parameter.

```
struct AcSAPfinish {
    int     acf_reason;

    int     acf_ninfo;
    char    acf_info[NACDATA];
};
```

The elements of this structure are:

`acf_reason:` the reason for the release request, one of:

Value	Reason
ACF_NORMAL	normal
ACF_URGENT	urgent
ACF_UNDEFINED	undefined

`acf_info/acf_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 ACFFREE macro when no longer referenced. The ACFFREE macro, behaves as if it was defined as:

```
void    ACFFREE (acf)
struct AcSAPfinish *acf;
```

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

Otherwise, if the call fails, the aci parameter contains information pertaining to the failure.

2.3.2 Abort Indication

Upon receiving a P-U-ABORT.INDICATION or P-P-ABORT.INDICATION event, the routine AcABORTser should be called to map this into a A-ABORT.INDICATION or A-P-ABORT.INDICATION event.

```
int     AcABORTser (sd, pa, aci)
int     sd;
struct PSAPabort *pa;
struct AcSAPindication *aci;
```

The parameters to this procedure:

sd: the association-descriptor;

pa: a pointer to an PSAPabort structure, containing information on the presentation-level abort; and,

aci: a pointer to an AcSAPindication structure.

If the call to AcABORTser is successful, then the aci parameter is updated to reflect information on the abort indication. Otherwise, if the call fails, the aci parameter contains information pertaining to the failure.

2.4 Select Facility

Ideally, the *select*(2) system call should be used on all descriptors, regardless of the level of abstraction (e.g., association-descriptor, presentation-descriptor, and so on). The routines with a `SelectMask` suffix, such as `PSelectMask` are supplied to determine if there are already queued events. These routines should always be called prior to using the select facility of the kernel. Sadly, some networking subsystems used by the software do not support *select*(2). To provide a consistent interface, the `xselect` routine should be used instead of *select*(2).

```
int     xselect (nfds, rfds, wfds, efds, secs)
int     nfds;
fd_set *rfds,
       *wfds,
       *efds,
        secs;
```

The parameters to this procedure are:

`nfds`: the number of descriptors present in the masks (actually the `width` of descriptors from zero to the highest meaningful descriptor);

`rfds/wfds/efds`: the locations of masks interested in read, write, and exception events; and,

`secs`: the maximum number of seconds to wait for an event (a value of `NOTOK` indicates that the call should block indefinitely, whereas a value of `OK` indicates that the call should not block at all, e.g., a polling action).

Unlike most routines, the `xselect` routine returns one of several values: `NOTOK` (on failure), `OK` (if no events occurred within the time limit), or, the number of descriptors on which events occurred (the masks are updated appropriately).

By default, unless the application explicitly ignores `SIGINT`, when the user types a CTRL-C this will interrupt the call to `xselect`. If you make calls directly to `xselect`, you can disable this behavior by setting

```
extern int xselect_blocking_on_intr;
```

to a non-zero value. (When reading from an association-descriptor, this behavior is automatically disabled.)

2.5 Generic Server Dispatch

Ideally, one should write a server which can operate in one of two modes:

- Each incoming connection results in *tsapd*(8c) invoking another instance of the server; or,
- All incoming connections are given to exactly one instance of the server (the server is invoked without arguments during system initialization).

The choice as to which mode is used is made by the system administrator. By default, the first mode, termed the *dynamic* approach, is used. The second mode, termed the *static* approach, is described in Section 10.2 on page 162.

There are several ways in which this dual-approach scheme can be realized. One such implementation is based on the routine `isodeserver`.

```
int     isodeserver (argc, argv, aei, initfnx, workfnx,
                losefnx, td)
int     argc;
char  **argv;
AEI     aei;
IFP     initfnx,
        workfnx,
        losefnx;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`argc`: the length of the argument vector which the program was invoked with;

`argv`: the argument vector which the program was invoked with;

`aei`: the information on the application-entity offering the service;

`initfnx`: the address of an event-handler routine to be invoked when a new connection arrives;

`workfnx`: the address of an event-handler routine to be invoked when activity occurs on an association;

`losefnx`: the address of an event-handler routine to be invoked if `TNetAccept` (described in Section 4.7) fails and,

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

If the call is successful, then the program may terminate immediately, as no work remains to be done (in the case of the single instance mode, `isodeserver` returns only on error). Otherwise, the `td` parameter indicates the reason for failure (consult Section 4.2.2 on page 108 of *Volume Two*).

When an event associated with a new connection occurs, the event-handler routine is invoked with two parameters:

```
(*initfnx) (vecp, vec);
int     vecp;
char  **vec;
```

The parameters are:

`vecp`: the length of the argument vector; and,

`vec`: the argument vector.

The event-handler should then call `AcInit` with these parameters to achieve an A-ASSOCIATION.INDICATION event. If `AcInit` is successful, the event-handler should decide if it wishes to honor the association and then call `AcAssocResponse` with the appropriate parameters. The event-handler should return the assocation-descriptor if it accepted the association. Otherwise (or upon any errors), it should return `NOTOK`.

When an activity associated with an association occurs, the event-handler routine is invoked with one parameter:

```
(*workfnx) (fd);
int     fd;
```

The parameter is:

`fd:` the association-descriptor of the association.

The event-handler should then call the appropriate routine to read the next event from the association (e.g., `RoWaitRequest` if remote operations are being used). The event-handler should return `NOTOK` if the association is terminated or aborted, or `OK` otherwise.

The `isodeserver` routine uses the `TNetAccept` routine to wait for the next event on existing associations and for new connections. It is possible, though unlikely for a failure to occur during this operation. In this event, the event-handler routine is invoked with one parameter:

```
(*losefnx) (td);
struct TSAPdisconnect *td;
```

The parameter is:

`td:` a pointer to an `TSAPdisconnect` structure updated by `TNetAccept`.

The event-handler should handle the error however it deems proper (usually by logging it and then returning).

An example of this facility is presented in Section 2.9 below.

Another interface to this approach is provided for servers that may be engaged in other operations besides answering incoming calls (e.g., initiating outgoing calls too). This interface provides indentical functionality to the `isodeserver` interface, but allows access to other events.

The interface is provided by two calls. The first is `iserver_init`.

```
int     iserver_init (argc, argv, aei, initfnx, td)
int     argc;
char  **argv;
AEI     aei;
IFP     initfnx;
struct TSAPdisconnect *td;
```

The parameters to this procedure are similar in part to the `isodeserver` procedure:

`argc:` the length of the argument vector which the program was invoked with;

`argv`: the argument vector which the program was invoked with;

`aei`: the information on the application-entity offering the service;

`initfnx`: the address of an event-handler routine to be invoked when a new connection arrives, and

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

This procedure registers a listener for the address and may may call the `initfnx` if the server is running in dynamic mode. If the call fails the `td` parameter will indicate the reason for failure.

One this function has been called, the `iserver_wait` procedure should be called at regular intervals to handle incoming events, typically in a loop of some kind. It is called as follows:

```
int     iserver_wait (initfnx, workfnx, losefnx, nfds,
                      nfds, rfds, wfds, efds, secs, td)
IFP     initfnx,
        workfnx,
        losefnx;
int     nfds;
fd_set *rfds,
       *wfds,
       *efds;
int     secs;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`initfnx`: the address of an event-handler routine to be invoked when a new connection arrives;

`workfnx`: the address of an event-handler routine to be invoked when activity occurs on an association;

`losefnx`: the address of an event-handler routine to be invoked if `TNetAccept` (described in Section 4.7) fails;

`nfds/rfds/wfds/efds:` additional association-descriptors/file descriptors to await for activity on;

`secs:` the maximum number of seconds to wait for activity (a value of `NOTOK` indicates that the call should block indefinitely, whereas a value of `OK` indicates that the call should not block at all, e.g., a polling action); and

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

This routine calls the `TNetAccept` routine to wait for incoming calls. It will call the procedures `initfnx`, `workfnx` and `losefnx` in an indetical way to the `isodeserver`. The routine returns on a number of conditions.

- If one of the supplied association/file descriptors registers activity the procedure returns with the mask updated.
- If an incoming association occurs. It should be accepted or rejected by the `initfnx` then the procedure will return.
- If some activity happens on one of the accepted calls. This is handled by the `workfnx` then the procedure will return.
- If the time given runs out, the procedure will return.

An outline usage of these procedures might be something like:

```
if (iserver_init (argc, argv, aei, ros_init, td) == NOTOK)
    /* error */
for (;;) {
    /* set up descriptors */
    switch (iserver_wait (ros_init, ros_work, ros_lose, nfds,
                &rfds, &wfds, NULLFD, timeout, td)) {
        case DONE:
            exit (0);

        case NOTOK:
            /* error */

        case OK:
```