```
struct PSAPaddr *calling,
                *called;
int     prequirements,
        srequirements,
        settings,
        ndata;
long isn;
struct PSAPctxlist *ctxlist;
OID     defctxname;
struct SSAPref *ref;
PE    *data;
struct QOStype *qos;
struct PSAPconnect *pc;
struct PSAPindication *pi;
```

The parameters to this procedure are:

`calling`: the PSAP address of the initiator (use the manifest constant verb"NULLPA" if this is not unimportant);

`called`: the PSAP address of the responder;

`ctxlist`: the list of proposed presentation contexts (only the `pc_id`, `pc_asn`, and optionally the `pc_atn` elements should be filled in);

`defctxname`: the proposed default contexts;

`prequirements`: the presentation requirements;

`srequirements`: the session requirements;

`isn`: the initial serial-number;

`settings`: the initial token settings;

`ref`: the connection identifier used by the initiator (consult page 52 for a description of the `SSAPref` structure);

`data/ndata`: an array of initial data (and the number of elements in that array, consult the warning on page 35);

`qos`: the quality of service on the connection (see Section 4.6.2);

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

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

If the call to `PConnRequest` is successful (a successful return corresponds to a P-CONNECT.CONFIRMATION event), then information is returned in the `pc` parameter, which is a pointer to a `PSAPconnect` structure.

```
struct PSAPconnect {
    int     pc_sd;

    struct PSAPaddr pc_responding;

    struct PSAPctxlsit pc_ctxlist;

    int     pc_defctxresult;

    int     pc_prequirements;

    int     pc_srequirements;
    int     pc_settings;
    int     pc_please;
    long    pc_isn;

    struct SSAPref pc_connect;

    int     pc_ssdusize;

    struct QOStype pc_qos;
    int     pc_result;

    struct SSAPref  pc_connect;

#define PC_SIZE         512
    int     pc_cc;
    char    pc_data[PC_SIZE];
};
```

The elements of this structure are:

`pc_sd`: the presentation-descriptor to be used to reference this connection;

`pc_responding`: the responding peer's address (which is the same as the `called` parameter given to `SConnRequest`);

`pc_ctxlist`: the (negotiated) list of presentation contexts;

`pc_defctxresult`: the response to the proposed default context;

`pc_prequirements`: the (negotiated) presentation requirements;

`pc_srequirements`: the (negotiated) session requirements;

`pc_settings`: the (negotiated) initial token settings;

`pc_please`: the tokens which the responder wants to own (if any);

`pc_isn`: the (negotiated) initial serial-number;

`pc_connect`: the connection identifier used by the responder (consult page 52 for a description of the `SSAPref` structure);

`pc_ssdusize`: the largest atomic SSDU size that can be used on the connection (see the note on page 45);

`pc_qos`: the quality of service on the connection (see Section 4.6.2); and,

`pc_result`: the connection response;

`pc_info/pc_ninfo`: an array of initial data, and the number of elements in that array.

If the call to `PConnRequest` is successful, and the `pc_result` element is set to `PC_ACCEPT`, then connection establishment has completed and the users of the presentation service now operate as symmetric peers. If the call is successful, but the `pc_result` element is not `PC_ACCEPT`, then the connection has been rejected; consult Table 2.1 to determine the reason (further information can be found in the `pi` parameter). Otherwise, if the call fails then the connection is not established and the `PSAPabort` structure of the `PSAPindication` discriminated union has been updated.

Note that the `pc_info` element is allocated via *malloc*(3) and should be released using the `PCFREE` macro when no longer referenced. The `PCFREE` macro behaves as if it was defined as:

```
void    PCFREE (pc)
struct PSAPconnect *pc;
```

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

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

```
int     PAsynConnRequest (calling, called, ctxlist,
                defctxname, prequirements, srequirements,
                isn, settings, ref, data, ndata, qos, pc,
                pi, async)
struct PSAPaddr *calling,
                *called;
int     prequirements,
        srequirements,
        settings,
        ndata,
        async;
long isn;
struct PSAPctxlist *ctxlist;
OID     defctxname;
struct SSAPref *ref;
PE    *data;
struct QOStype *qos;
struct PSAPconnect *pc;
struct PSAPindication *pi;
```

The additional parameter to this procedure is:

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

If the `async` parameter is non-zero, then `PAsynConnRequest` returns one of four values: `NOTOK`, which indicates that the connection request failed; `DONE`, which indicates that the connection 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 `PConnRequest` are employed (i.e., a `NOTOK` return from `PAsynConnRequest` is equivalent to a `NOTOK` return from `PConnRequest`, and, a `DONE` return from `PAsynConnRequest` is equivalent to a `OK` return from `PConnRequest`).

In the final case, when either `CONNECTING_1` or `CONNECTING_2` is returned, only the `pc_sd` element of the `pc` parameter has been updated; it reflects the presentation-descriptor to be used to reference this connection.

To determine when the connection attempt has been completed, the routine `xselect` (consult Section 2.4 of *Volume One*) should be used after calling `PSelectMask`. In order to determine if the connection attempt was successful, the routine `PAsynRetryRequest` is called:

```
int     PAsynRetryRequest (sd, pc, pi)
int     sd;
struct PSAPconnect *pc;
struct PSAPindication  *pi;
```

The parameters to this procedure are:

`sd`: the presentation-descriptor;

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

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

Again, one of three values are returned: `NOTOK`, which indicates that the connection request failed; `DONE`, which indicates that the connection 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 for information on how to make efficient use of the asynchronous connection facility.

2.4 Data Transfer

Once the connection has been established, a presentation-descriptor is used to reference the connection. This is usually the first parameter given to any of the remaining routines in the *libpsap2*(3n) library. Further, the last parameter is usually a pointer to a `PSAPindication` structure (as described on page 23). If a call to one of these routines fails, then the structure is updated. Consult the `PSAPabort` element of the `PSAPindication` structure. If the `pa_reason` element of the `PSAPabort` structure is associated with a fatal error, then the connection is closed. That is, a P-P-ABORT.INDICATION event has occurred. The `PC_FATAL` macro can be used to determine this.

```
int     PC_FATAL (r)
int     r;
```

The most common interface error to occur is `PC_OPERATION` which usually indicates that either the user is lacking ownership of a session token to perform an operation, or that a session requirement required by the operation was not negotiated during connection establishment. For protocol purists, the `PC_OFFICIAL` macro can be used to determine if the error is an "official" error as defined by the specification, or an "unofficial" error used by the implementation.

```
int     PC_OFFICIAL (r)
int     r;
```

All of the remaining routines in the *libpsap2*(3n) library are identical to their counterparts in the *libssap*(3n) library, with these exceptions:

- The final parameter to each routine is a pointer to a `PSAPindication` structure, rather than a `SSAPindication` structure.
- Any user data components are specified as an array of presentation elements (and the number of elements in that array), rather than the base of a character array (and the number of octets to be sent). Note that any data to be sent should have the `pe_context` element set to the desired presentation context. By default, presentation elements are initialized with the default context (as represented by the manifest constant `PE_DFLT_CTX`).

- Asynchronous event handlers are called with pointers to `PSAP` structures, rather tha `SSAP` structures.
- With any indication which occurs, it is important to free any data which might have been allocated. The structures and corresponding macros are:

Macro	Structure Pointer
`PXFREE`	struct PSAPdata *
`PTFREE`	struct PSAPtoken *
`PNFREE`	struct PSAPsync *
`PVFREE`	struct PSAPactivity *
`PPFREE`	struct PSAPreport *
`PFFREE`	struct PSAPfinish *
`PRFREE`	struct PSAPrelease *
`PAFREE`	struct PSAPabort *

Note that these free the user data referenced by the indication structures, and not the structures themselves.

2.4.1 Restrictions on User Data

To quote the [ISO88a] specification:

> **NOTE:** For all services which carry user data, excluding P-DATA and P-TYPED-DATA, it may not be possible to exchange user data, dependent on the user data length limitation supported by the underlying session services.

2.5 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `pi` parameter given to the routine. The `pi_abort` element of the `PSAPindication` structure contains a `PSAPabort` structure detailing the reason for the failure. The `pa_reason` element of this latter structure can be given as a parameter to the routine `PErrString` which returns a null-terminated diagnostic string.

```
char   *PErrString (c)
int     c;
```

2.6 Compiling and Loading

Programs using the *libpsap2*(3n) library should include `<isode/psap2.h>`. The programs should also be loaded with `-lpsap2`.

2.7 An Example

Let's consider how one might construct a source entity that resides on the PSAP. This entity will use a synchronous interface.

There are two parts to the program: initialization and data transfer; release will occur when the standard input has been exhausted. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure 2.2, the initialization steps for the source entity, including the outer *C* wrapper, is shown. First, a lookup is done in the ISO services database, and the `PSAPaddr` is initialized. The `SSAPref` is initialized using the routine `addr2ref`. This routine takes a string and returns a pointer to a `SSAPref` structure which has been initialized to contain the string and the current UTC time. Next, for each token associated with the session requirements, initial ownership of that token is claimed. Finally, the call to `PConnRequest` is made. If the call is successful, a check is made to see if the remote user accepted the connection. If so, the presentation-descriptor is captured, along with the negotiated requirements and initial token settings.

In Figure 2.3 on page 39, the data transfer loop is realized. The source entity reads a line from the standard input, and then queues that line for sending to the remote side. When an end-of-file occurs on the standard input, the source entity requests release and then gracefully terminates. Although no checking is done in this example, for the calls to `PDataRequest` and `PRelRequest`, on failure a check for the operational error `PC_OPERATION` should be made. For `PDataRequest`, this would occur when the data token was not owned by the user; for `PRelRequest`, this would occur when the release token was not owned by the user.

```
#include <stdio.h>
#include <isode/psap2.h>
#include <isode/isoservent.h>


static int requirements = SR_HALFDUPLEX | SR_NEGOTIATED;

static int owned = 0;


/* ARGSUSED */

main (argc, argv, envp)
int       argc;
char  **argv,
     **envp;
{
   int     sd,
             settings;
   char    buffer[BUFSIZ];
   register struct PSAPaddr   *pz;
   register struct SSAPref *sf;
   struct PSAPconnect  pcs;
   register struct PSAPconnect *pc = &pcs;
   struct PSAPrelease  prs;
   register struct PSAPrelease *pr = &prs;
   struct PSAPindication   pis;
   register struct PSAPindication *pi = &pis;
   register struct PSAPabort *pa = &pi -> pi_abort;
   register struct isoservent *is;

   if ((is = getisoserventbyname ("sink", "psap")) == NULL)
         error ("psap/sink:  unknown provider/entity pair");
   if (argc != 2)
         error ("usage:  source \"host\"");

...
```

Figure 2.2: Initializing the PSAP source entity

```
...

    if ((pz = is2paddr (argv[1], NULLCP, is)) == NULLPA)
            error ("address translation failed");
    sf = addr2ref (PLocalHostName ());

    settings = 0;
#define dotoken(requires,shift,bit,type) \
{ \
    if (requirements & requires) \                                    10
            settings |= ST_INIT_VALUE << shift; \
}
    dotokens ();
#undef dotoken

    if (PConnRequest (NULLSA, pz, NULLPC, NULLOID, 0, requirements,
                SERIAL_NONE, settings, sf, NULLPEP, 0, NULLQOS, pc, pi) == NOTOK)
            error ("P-CONNECT.REQUEST: %s", PErrString (pa -> pa_reason));
    if (pc -> pc_result != PC_ACCEPT)
            error ("connection rejected by sink:  %s",                20
                        PErrString (pc -> pc_result));

    sd = pc -> pc_sd;
    requirements = pc -> pc_requirements;

#define dotoken(requires,shift,bit,type) \
{ \
    if (requirements & requires) \
            if ((pc -> pc_settings & (ST_MASK << shift)) \
                    == ST_INIT_VALUE) \                               30
                owned |= bit; \
}
            dotokens ();
#undef dotoken

    PCFREE (pc);

...
```

Figure 2.2: Initializing the PSAP source entity (continued)