```
IFP     rfx,
        wfx;
```

The parameters to this procedure are:

`ps:` the presentation stream;

`fd:` the file-descriptor;

`size:` the maximum datagram size; and,

`rfx/wfx:` routines to read and write datagrams.

After the setup routine successfully returns (by returning the manifest constant `OK`), the presentation stream is ready for reading or writing.

5.1.2 Stream I/O

Low-level I/O is done from/to the stream by the macros `ps_read` and `ps_write`, which behave as if they were defined as:

```
int     ps_read (ps, data, cc, inline)
PS      ps;
char   *data;
int     cc,
        inline;

int     ps_write (ps, data, cc, inline)
PS      ps;
char   *data;
int     cc,
        inline;
```

The parameters to both of these macros are the same:

`ps:` the presentation stream;

`data:` the address of a character buffer;

`cc:` the number of characters to read/write from/to the buffer; and,

`inline:` a magic argument, always use `0` unless you know what you're doing.

These both call an internal routine, `ps_io`, which switches to the object-specific read or write routine as appropriate. The `ps_io` procedure will call the object-specific routines as many times as required to read/write the full number of `cc` bytes from/to the `data` buffer.

5.1.3 Deleting a Stream

The routine `ps_free` is used to close and deallocate a presentation stream.

```
void    ps_free (ps)
PS      ps;
```

It takes a single parameter, a pointer to the presentation stream to be freed. This routine first calls the routine specified by the `ps_closeP` element in the `PStream` structure (if any). It then frees the structure itself.

5.1.4 Implementing Other Abstractions

Let us briefly consider the internal protocol and uniform interface used in the implementation of presentation streams.

The initialization routine given as an argument to `ps_alloc` typically initializes only the

```
ps_primeP
ps_readP
ps_writeP
ps_flushP
ps_closeP
```

elements in the `PStream` structure.

The setup routine is entirely dependent on the particular object used to realize the I/O abstraction for the presentation stream. In most cases, it allocates a structure of its own and sets the `ps_addr` element to the address of the structure.

The `ps_readP` and `ps_writeP` elements of the `PStream` structure are used by the `ps_io` routine as required for reading and writing.

```
int     ps_io (ps, io, data, cc, inline)
PS      ps;
```

```
int   (*io) ();
char   *data;
int     cc,
```

The parameters to these routines are identical to those for their counterpart macros, `ps_read` and `ps_write`, with one exception:

`io:` the address of an integer-valued function which does the actual reading or writing. It is invoked as:

```
int     n = (*io) (ps, data, len, inline);
```

where a return value of `NOTOK` indicates an error occurred (the routine should set the `ps_errno` element of the presentation stream); `OK` indicates that the end-of-file has been read; and, any other value is the number of bytes actually transferred (which should be greater than `0` but not greater than `len`).

For some packet-oriented applications, it may be desirable to do a single "read from the network" before reading the components of a presentation element. The routine pointed to by the `ps_primeP` element of the `PStream` structure is called by `ps2pe` (described momentarily) before any parts of the entire presentation element has been read. The routine is invoked as:

```
(*ps -> ps_primeP) (ps, waiting)
PS      ps;
int waiting;
```

The routine should return `NOTOK` if an error occurred (setting the `ps_errno` element of the presentation stream in the process); otherwise, if data is already queued and `waiting` is non-zero, then `DONE` is returned. otherwise `OK` is returned. The routine `ps_prime` may be called to explicitly "prime the pump" associated with a presentation element:

```
int     ps_prime (ps)
PS      ps;
```

This routine returns `OK` on success, or `NOTOK` on failure.

In order to improve efficiency, it may be desirable to have `ps_writeP` buffer output. The routine pointed to by the `ps_flushP` element of the `PStream` structure is called by `pe2ps` (described momentarily) after the entire presentation element has been written. The routine is invoked as:

```
(*ps -> ps_flushP) (ps)
PS      ps;
```

The routine should return NOTOK if an error occurred (setting the ps_errno element of the presentation stream in the process); or OK if everything was fine. The routine ps_flush may be called to explicitly flush any buffers associated with a presentation element:

```
int     ps_flush (ps)
PS      ps;
```

This routine returns OK on success, or NOTOK on failure.

Finally, the ps_closeP element of the PStream structure is used by the ps_free routine to release any resources which the setup routine may have allocated. For example, if the setup routine allocated a structure and set the ps_addr element of the presentation stream to point to that structure, then the function pointed to by the ps_closeP element should free that structure, and, as a matter of good programming practice, set ps_addr to NULL.

5.2 Presentation Stream I/O

The routine ps2pe can be used to read the next presentation element from a presentation stream. This routine returns a pointer to the presentation element or the manifest constant NULLPE on error. (The typedef PE is a pointer to a structure containing a presentation element; presentation elements are described more fully later.)

```
PE      ps2pe (ps)
PS      ps;
```

Similarly, the routine pe2ps can be used to write a presentation element at the end of the presentation stream, returning OK if all went well, or NOTOK otherwise.

```
int     pe2ps (ps, pe)
PS      ps;
PE      pe;
```

On errors with either routine, the `ps_errno` element of the `PStream` structure can be consulted to see what happened (see Table 5.1 on page 116).

When writing to a presentation stream, the variable `ps_len_strategy` controls how lengthy data structures are represented by determining when the "indefinite form" is used to encode the length of the data structure.

```
int     ps_len_strategy;
```

If this variable is equal to `PS_LEN_SPAG` (the default), then the indefinite form is used whenever the length field of the presentation element cannot be represented in one octet. If the value instead is `PS_LEN_INDF`, then the indefinite form is used regardless of the length of the presentation element. Otherwise, if the value is `PS_LEN_LONG`, then the indefinite form is never used.

The routine `ps_get_abs` can be used to determine the total number of octets that will be required to represent the presentation element when written to a presentation stream. This is useful for buffer management purposes.

```
int     ps_get_abs (pe)
PE      pe;
```

5.2.1 Debugging

For debugging purposes, instead of treating a presentation stream as a binary object, the routines `pl2pe` and `pe2pl` can be used.

```
PE      pl2pe (ps)
PS      ps;

int     pe2pl (ps, pe)
PS      ps;
PE      pe;
```

These translate between presentation *lists* and presentation elements. A presentation list is identical to a presentation stream, but instead of using a binary representation, a list uses an ASCII text representation with a simple LISP-like syntax.

5.3 Presentation Elements

A presentation element is an object which is used to represent a data structure in a machine-independent form. The `PElement` structure contains several elements, only the most interesting are described here:

`pe_errno`: the latest error to occur on the presentation element (codes are listed in Table 5.2);

`pe_context`: the presentation context to which the element belongs (consult Section 2.3.1 on page 18 of *Volume Two*);

`pe_class`: the class of this presentation element (i.e., one of *universal*, *application-wide*, *context-specific*, or *private*);

`pe_form`: the form taken by this presentation element (i.e., *primitive*, or *constructed*);

`pe_id`: the class-specific code identifying the type of this presentation element;

`pe_offset`: the offset of this presentation element in a sequence; and,

`pe_next`: a pointer to the next presentation element in a sequence.

As mentioned earlier, the typedef `PE` is a pointer to an `PElement` structure. With the exception of the `pe_errno` element, the elements of the `PElement` structure are largely uninteresting to the user of the *libpsap*(3) library. The element `pe_errno` can be given as a parameter to the routine `pe_error` which returns a null-terminated diagnostic string.

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

Once a presentation stream has been initialized and elements are being read, there are several routines which can be used to translate between the machine-independent representation of the element and machine-specific objects such as integers, strings, and the like. It is extremely important that programs use these routines to perform the translation between objects. They have been carefully coded to present a simple, uniform interface between machine-specifics and the machine-independent encoding protocol.

PE_ERR_NONE	No error
PE_ERR_OVER	Overflow
PE_ERR_NMEM	Out of memory
PE_ERR_BIT	No such bit
PE_ERR_UTCT	Malformed universal timestring
PE_ERR_GENT	Malformed generalized timestring
PE_ERR_MBER	No such member
PE_ERR_PRIM	Not a primitive form
PE_ERR_CONS	Not a constructor form
PE_ERR_TYPE	Class/ID mismatch in constructor
PE_ERR_OID	Malformed object identifier
PE_ERR_BITS	Malformed bitstring

Table 5.2: Presentation Element Failure Codes

5.3.1 Creating an Element

A `PElement` structure is created by calling the procedure `pe_alloc`.

```
PE      pe_alloc (class, form, id)
PElementClass class;
PElementForm form;
PElementID id;
```

This procedure takes three parameters:

`class`: the class of this presentation element. The codes are:

PE_CLASS_UNIV	Universal
PE_CLASS_APPL	Application-wide
PE_CLASS_CONT	Context-specific
PE_CLASS_PRIV	Private-use

`form`: the form of this presentation element The codes are:

PE_FORM_PRIM	Primitive
PE_FORM_CONS	Constructor

`id`: the class-specific code identifying the type of this presentation element (codes for the "universal" class are listed in Table 5.3).

5.3.2 Deleting an Element

The routine `pe_free` is used to deallocate a presentation element.

```
void    pe_free (pe)
PE      pe;
```

> **NOTE:** When using `pe_free` on a presentation element, care must be taken to remove any references to that presentation element in other structures. For example, if you have a sequence containing a sequence, and you free the child sequence, be sure to zero-out the parent's pointer to the child, otherwise subsequent calls using the parent will go romping through hyperspace. See `pe_extract` and `pe_expunge` below.

5.3.3 Primitive Manipulation of Elements

Two presentation elements can be compared with `pe_cmp`, which takes two pointers to `PElement` structures as arguments, and returns `0` if the two structures are identical, `1` otherwise.

```
int     pe_cmp (p, q)
PE      p,
        q;
```

Further, an presentation element can be duplicated with `pe_cpy`. This routine takes a pointer to an `PElement` structure as an argument, and returns a pointer to a new `PElement` structure, or `NULLPE` on error.

```
PE      pe_cpy (pe)
PE      pe;
```

If a presentation element, `pe`, has a descendant `r` (which appears below `pe` exactly once), then `pe_extract` can be used to destroy the relationship between the two presentation elements without freeing either element.

Internal Use	
PE_UNIV_EOC	End-of-contents
Built-in Types	
PE_PRIM_BOOL	Boolean
PE_PRIM_INT	Integer
PE_PRIM_BITS	Bitstring
PE_PRIM_OCTS	Octetstring
PE_PRIM_NULL	Null
PE_PRIM_OID	Object Identifier
PE_PRIM_ODE	Object Descriptor
PE_CONS_EXTN	External
PE_CONS_SEQ	Sequence
PE_CONS_SET	Set
Defined Types	
PE_DEFN_IA5S	IA5 String
PE_DEFN_NUMS	Numeric String
PE_DEFN_PRTS	Printable String
PE_DEFN_T61S	T.61 String
PE_DEFN_VTXS	Videotex String
PE_DEFN_GENT	Generalized Time
PE_DEFN_UTCT	UTC Time
PE_DEFN_GFXS	Graphics String
PE_DEFN_VISS	Visual String
PE_DEFN_GENS	General String

Table 5.3: Presentation Element Identifiers

```
int     pe_extract (pe, r)
PE      pe,
        r;
```

This routine returns non-zero if `r` was descended from `pe`, otherwise it returns zero. The `pe_expunge` routine is similar to `pe_extract`, except that it always deallocates the parent element and returns the child element found, if any.

```
PE      pe_expunge (pe, r)
PE      pe,
        r;
```

These two routines are provided primarily for efficiency considerations. They are extremely fragile. Do not use them unless you know what you're doing.

Finally, the `pe_pullup` routine can be used to "pull-up" a constructor type into a primitive.

```
int     pe_pullup (pe)
PE      pe;
```

This routine returns `OK` on success; on failure it turns `NOTOK`, usually due to a memory allocation failure.

5.4 Presentation Element Transformations

We now discuss how machine-specific objects can be encoded in a machine-independent fashion and vice-versa. Routines of the form `XXX2prim` all return a pointer to a presentation element on success, or `NULLPE` on failure (usually due to a failure in the memory allocator). Routines of the form `prim2XXX` all return an `XXX`-valued object on success, or `NOTOK` or `NULLxxx` on failure (depending on the object the routine normally returns). On failure, the `pe_errno` element of the presentation element can be examined to determine the reason for failure.

5.4.1 Boolean

A *boolean* is an integer taking on the values `0` or `1`. The routine `prim2flag` takes a pointer to an presentation element and returns the boolean value encoded therein.