```
char   *time2str (ut, generalized)
UTC     u;
int     generalized;
```

Both call the routine `prim2time`:

```
UTC     prim2time (pe, generalized)
PE      pe;
int     generalized;
```

The routines `str2utct` and `str2gent` perform the inverse operations:

```
UTC     str2utct (cp, len)
char   *cp;
int     len;

UTC     str2gent (cp, len)
char   *cp;
int     len;
```

Each take a character-pointer and a length-indicator as arguments and return a `UTCtime` structure on success. Otherwise, the manifest constant `NULLUTC` is returned.

There are also some utility routines to aid in converting between `UTCtime` and UNIX `tm` time structures. The routine `gtime` is the inverse of *gmtime*(3), it takes a time structure and returns a long-valued number.

```
long    gtime (tm)
struct tm *tm;
```

The routine `ut2tm` returns a pointer to a static time structure which has been initialized by its `UTC` argument.

```
struct tm *ut2tm (ut)
UTC     ut;
```

Finally, the routine `tm2ut` performs the inverse operation, initializing a `UTC` argument by using a time structure argument.

```
void     tm2ut (tm, ut)
struct tm *tm;
UTC     ut;
```

5.4.8 Sets and Sequences

Two list disciplines are implemented: *sets*, in which each member is distinguished by a unique identifier; and, *sequences*, in which each element is distinguished by its offset from the head of the list. It should be noted that these definitions of sets and sequences do not exactly match those defined in ASN.1. In particular, the ASN.1 structure `SET␣OF` must be defined as a type `SET`, but be built up with the `seq_add` construct, as these elements all have the same type. On both types of lists, the macro `first_member` returns the first member in the list, while `next_member` returns the next member. These macros behave as if they were defined as:

```
PE      first_member (head)
PE      head;

PE      next_member (head, member)
PE      head,
        member;
```

There are three operations on sets:

`set_add:` adds a new member to the set;

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

The routine `set_addon` is an efficient version of `set_add` that is used when adding several consecutive members to a set.

`set_del:` removes the identified member from the set;

```
int     set_del (pe, class, id)
PE      pe;
PElementClass class;
PElementID id;
```

and,

`set_find:` locates the identified member.

```
PE      set_find (pe, class, id)
PE      pe;
PElementClass class;
PElementID id;
```

The routines `prim2set` and `set2prim` are used to translate between presentation elements and sets.

```
PE      prim2set (pe)
PE      pe;

PE      set2prim (pe)
PE      pe;
```

In Figure 5.1, a convenient way of stepping through all the members of a set is presented.

There are three operations on sequences:

`seq_add:` adds a new member `r` to the sequence `pe`, at the given offset, an offset of `-1` will add the element to the end of the sequence;

```
int     seq_add (pe, r, offset)
PE      pe,
        r;
int     offset;
```

The routine `seq_addon` is an efficient version of `seq_add` that is used when adding several consecutive elements to a set.

`seq_del:` removes the identified member from the sequence;

```
int     seq_del (pe, offset)
PE      pe;
int     offset;
```

and,

`seq_find:` locates the identified element.

```
PE      seq_find (pe, offset)
PE      pe;
int     offset;
```

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

...

register struct PE pe, p;

...

for (p = first_member (pe); p; p = next_member (pe, p))
    switch (PE_ID (p -> pe_class, p -> pe_id)) {                    10
        case PE_ID (PE_CLASS_UNIV, PE_DEFN_IA5S):
            ...

        case PE_ID (PE_CLASS_UNIV, PE_DEFN_T61S):
            ...

        ...
    }

...                                                                 20
```

Figure 5.1: Stepping through a Sequence

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

...

register struct PE pe, p;

...

for (p = first_member (pe); p; p = next_member (pe, p))
    switch (p -> pe_offset) {
        case 0:
            ...

        case 1:
            ...

        ...
    }

...
```

Figure 5.2: Stepping through a Set

The routines `prim2seq` and `seq2prim` are used to translate between presentation elements and sequences.

```
PE      prim2seq (pe)
PE      pe;

PE      seq2prim (pe)
PE      pe;
```

In Figure 5.2, a convenient way of stepping through all the members of a sequence is presented.

5.5 Inline CONStructors

Please read the following important message.

> **NOTE:** The facilities described in this section are to be used only as a last resort. They are provided for compatibility with less-rich systems.

When interfacing external software on top of various libraries, such as *librosap*(3n), the Remote Operations library, arguments which are expected to be `PElement`s may be available instead as the raw encoded octets. It is inefficent to convert this into a `PElement`, pass the structure to the library, which then just encodes it again. To avoid this behavior, the routine, `str2pe`, may be used:

```
PE      str2pe (s, len, advance, result)
char   *s;
int     len,
       *advance,
       *result;
```

The parameters to this procedure are:

`s/len:` the encoded string (and its length);

`advance:` a pointer to an integer-valued location indicating how far to advance the string to find the next ASN.1 object (use the manifest constant `NULLIP` if you aren't interested in this), and,

`result:` a pointer to an integer-valued location which indicates, if the call fails, the reason for the failure.

On success, `str2pe` returns a `PElement` called an *Inline CONStructor*. This is a special kind of `PElement` which really just contains a pointer to the original string. The underyling libraries now how to manipulate Inline CONStructors appropriately, e.g., when building an SSDU. On failure, `str2pe` returns the manifest constant `NULLPE` and the `result` parameter contains an error code listed in Table Table 5.1 on page 116.

For the inverse operation, the appropriate library must be informed that the ASN.1 objects associated with incoming indications should be returned

in Inline CONStructor form. This mechanism is still being refined, and is not currently documented.

However, the routine used by the libraries in order to unwrap an encoded string is available:

```
PE      qb2pe (qb, len, depth, result)
struct qbuf *qb
int     len,
        depth,
       *result;
```

The parameters to this procedure are:

`qb/len:` a doubly-linked list of `qbuf`s containing, e.g., an SSDU, and the total length of the data in the `qbuf`s;

`depth:` the level of unwrapping to permit; and,

`result:` a pointer to an integer-valued location which indicates, if the call fails, the reason for the failure.

On success, `qb2pe` returns a `PElement` which may ultimately contain Inline CONStructors. On failure, `qb2pe` returns the manifest constant `NULLPE` and the `result` parameter contains an error code listed in Table Table 5.1.

5.6 Compiling and Loading

Programs which manipulate presentation streams or presentation elements should include the header file `<isode/psap.h>`. These programs should also be loaded with `-lpsap` which contains the routines described in this chapter.

5.7 An Example

Let's consider how one might read a presentation stream from the standard input. The process is simple: we create a new presentation stream and then start the loop which reads from the stream until it is exhausted. This is shown in Figure 5.3. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

```
#include <stdio.h>
#include <isode/ppkt.h>

/* ARGSUSED */

main (argc, argv, envp)
int     argc;
char  **argv,
     **envp;
{                                                                    10
    int     len;
    char   *cp;
    register PS     ps;
    register PE     pe;

    if ((ps = ps_alloc (std_open)) == NULLPS)
        error ("ps_alloc");
    if (std_setup (ps, stdin) == NOTOK)
        error ("std_setup:  %s", ps_errno (ps -> ps_errno));
                                                                     20
    for (;;) {
        if ((pe = ps2pe (ps)) == NULLPE)
            if (ps -> ps_errno)
                error ("ps2pe:  %s", ps_errno (ps -> ps_errno));
            else
                break;              /* end-of-file */

/* process "pe" here... */

            pe_free (pe);                                            30
    }

    ps_free (ps);

    exit (0);
}
```

Figure 5.3: Stepping through a Presentation Stream

5.8 For Further Reading

The language for the Abstract Syntax Notation One (ASN.1) is specified in [ISO87a] and [CCI88e], while the encoding rules are defined in [ISO87b] and [CCI88e]. The older 1984 CCITT recommendations, containing both definitions, is [CCI84a], which is often referred to as X.409. The ASN.1 is a superset of the X.409.