There are two parts to the program: initialization and the request/reply loop. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure 4.1, the initialization steps for the generic responder, including the outer *C* wrapper, is shown. First, the RtSAP state is re-captured by calling `RtInit`. If this succeeds, then the association is authenticated and any command line arguments are parsed. Assuming that the responder is satisfied with the proposed association, it calls `RtOpenResponse` to accept the association. The `RoSetService` routine is called to set the underlying service to be used for remote operations. Finally, the main request/reply loop is realized. The responder calls `RoWaitRequest` to get the next event, and then calls `ros_indication` to decode that event.

Figure 3.2 on page 77 contains the definition of the `ros_indication` routine and associated routines. Since the reliable transfer service was used to establish the association, a different closing handler must be used. This is shown in Figure 4.2.

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


main (argc, argv, envp)
int       argc;
char  **argv,
     **envp;
{
    int       result,                                                    10
              sd;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;
    struct RtSAPstart   rtss;
    register struct RtSAPstart *rts = &rtss;
    struct AcSAPstart *acs = &rts -> rts_start;
    strut PSAPstart *ps = &acs -> acs_start;
    struct RtSAPindication  rtis;
    register struct RtSAPindication *rti = &rtis;                        20
    register struct RtSAPabort   *rta = &rti -> rti_abort;

    if (RtInit (argc, argv, rts, rti) == NOTOK)
        error ("initialization fails:   %s", RtErrString (rta -> rta_reason));

    sd = rts -> rts_sd;
    RTSFREE (rts);

/* read command line arguments here... */
                                                                         30
    if (RtOpenResponse (sd, ACS_ACCEPT, NULLOID, NULLAEI,
                    &ps -> ps_called, NULLPC, ps -> ps_defctxresult,
                    NULLPE, rti) == NOTOK)
        error ("RT-OPEN.RESPONSE: %s", RtErrString (rta -> rta_reason));

    if (RoSetService (sd, RoRtService, roi) == NOTOK)
        error ("RoSetService:   %s", RoErrString (rop -> rop_reason));

    for (;;)
        switch (result = RoWaitRequest (sd, NOTOK, roi)) {               40
            case NOTOK:
            case OK:
            case DONE:
                ros_indication (sd, roi);
                break;

            default:
                error ("unknown return from RoWaitRequest=%d", result);
        }
}                                                                        50
```

Figure 4.1: Initializing the generic RTS responder

```
...

static int  ros_finish (sd, rof)
int     sd;
struct AcSAPfinish *acf;
{
    struct RtSAPindication  rtis;
    register struct RtSAPabort   *rta = &rtis.rti_abort;

    if (RtCloseResponse (sd, ACR_NORMAL, NULLPE, &rtis) == NOTOK)                10
        error ("RT-CLOSE.RESPONSE: %s", RtErrString (rta -> rta_reason));

    ACFFREE (acf);

    exit (0);
}
```

Figure 4.2: Finalizing the generic RTS responder

4.6 For Further Reading

[CCI84b] is the CCITT recommendation describing the reliable transfer service. The draft CCITT work which assumes the use of new-style associations is defined in [CCI88a] and [CCI88b]. Similarly, the corresponding ISO work is [ISO88c] and [ISO88d].

Part III

Data Services

Chapter 5

Encoding of Data-Structures

The *libpsap*(3) library implements presentation syntax abstractions for the machine-independent exchange of data structures. There are two objects which are manipulated: *presentation elements*, which represent a particular, arbitrarily complex, data structure; and, *presentation streams*, which represent an I/O path of these data structures.

5.1 Presentation Streams

A presentation stream is an object, similar to a `FILE` object in *stdio*(3s), which is used to read and write *presentation elements*. The `PStream` structure contains several elements, only the most interesting are described here:

`ps_errno`: the latest error to occur on the stream (codes are listed in Table 5.1);

`ps_addr`: the bottom-specific pointer;

`ps_primeP`: the bottom-specific prime routine;

`ps_readP`: the bottom-specific read routine;

`ps_writeP`: the bottom-specific write routine; and,

`ps_flushP`: the bottom-specific flush routine; and,

`ps_closeP`: the bottom-specific close routine.

PS_ERR_NONE	No error
PS_ERR_OVERID	Overflow in ID
PS_ERR_OVERLEN	Overflow in length
PS_ERR_NMEM	Out of memory
PS_ERR_EOF	End of file
PS_ERR_EOFID	End of file reading extended ID
PS_ERR_EOFLEN	End of file reading extended length
PS_ERR_LEN	Length mismatch
PS_ERR_TRNC	Truncated
PS_ERR_INDF	Indefinite length in primitive form
PS_ERR_IO	I/O error
PS_ERR_XXX	Internal error code

Table 5.1: Presentation Stream Failure Codes

The typedef PS is a pointer to an PStream structure. The ps_errno element of the PStream structure can be given as a parameter to the routine ps_error which returns a null-terminated diagnostic string.

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

5.1.1 Creating a Stream

A PStream structure is created by calling the procedure ps_alloc, with the address of an integer-valued initialization routine.

```
PS      ps_alloc (init)
int   (*init) ();
```

The ps_alloc routine allocates a new structure and then calls the initialization routine passed as a parameter, which should initialize the elements of the structure. The initialization routine should return the manifest constant OK if all went well; otherwise it should return the manifest constant NOTOK on error, which results in ps_alloc freeing the newly allocated structure and then returning the value NULLPS.

Several standard initialization routines are available:

`std_open`: for presentation streams connected to *stdio*(3s) `FILE` objects;

```
int     std_open (ps)
PS      ps;
```

`str_open`: for presentation streams connected to string objects.

```
int     str_open (ps)
PS      ps;
```

`fdx_open`: for presentation streams connected to full-duplex file descriptors.

```
int     fdx_open (ps)
PS      ps;
```

and,

`dg_open`: for presentation streams connected to datagram-based file descriptors.

```
int     dg_open (ps)
PS      ps;
```

Presentation streams which have been initialized by these routines will automatically allocate additional resources when necessary, to the limits allowed by the operating system (e.g., repeated writes to a presentation stream connected to a string object will result in additional memory being allocated). In the current implementation, presentation streams which have been initialized by these routines are uni-directional. That is, the presentation stream may be used for reading, or writing, but not both.

After `ps_alloc` successfully returns, final initialization is performed by calling a setup routine, usually either

`std_setup`: for file objects.

```
int     std_setup (ps, fp)
PS      ps;
FILE   *fp;
```

The parameters to this procedure are:

`ps`: the presentation stream; and,

`fp`: a pointer to the `FILE` object to be bound to the presentation stream.

`str_setup`: for string objects.

```
int     str_setup (ps, cp, cc, inline)
PS      ps;
char   *cp;
int     cc,
inline;
```

The parameters to this procedure are:

`ps`: the presentation stream;

`cp`: a pointer to the string to be bound to the presentation stream (use the manifest constant `NULLCP` if the stream is to be written);

`cc`: the length of the string; and,

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

`fdx_setup`: for full-duplex file-descriptor objects.

```
int     fdx_setup (ps, fd)
PS      ps;
int     fd;
```

The parameters to this procedure are:

`ps`: the presentation stream; and,

`fd`: the file-descriptor.

`dg_setup`: for datagram objects.

```
int     dg_setup (ps, fd, size, rfx, wfx)
PS      ps;
int     fd,
        size;
```