```
#include <isode/rosap.h>
#include <isode/isoservent.h>

...

register struct SSAPaddr *sa;
struct RoSAPaddr roas;
register struct RoSAPaddr *roa = &roas;
register struct isoservent *is;

...

if ((is = getisoserventbyname ("directory", "rosap")) == NULL)
    error ("rosap/directory");

/* RemoteHost is the host we're interested in,
   e.g., "gremlin.nrtc.northrop.com" */

roa -> roa_port = is -> is_port;
if ((is = getisoserventbyname ("ros", "ssap")) == NULL)
    error ("ssap/ros");
if ((sa = is2saddr (RemoteHost, NULLCP, (struct isoservent *) is))
          == NULLSA)
    error ("address translation failed");
roa -> roa_addr = *sa;

...
```

Figure A.1: Constructing the RoSAP address for the Directory Services entity

```
int     RoInit (vecp, vec, ros, roi)
int     vecp;
char  **vec;
struct RoSAPstart *ros;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

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

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

If `RoInit` is successful, it returns information in the `ros` parameter, which is a pointer to a `RoSAPstart` structure.

```
struct RoSAPstart {
    int     ros_sd;

    struct RoSAPaddr ros_initiator;

    u_short ros_port;

    PE      ros_data;
};
```

The elements of this structure are:

`ros_sd`: the association-descriptor to be used to reference this association;

`ros_initiator`: the "unique identifier" of the initiator (containing the initiator's host address);

`ros_port`: the application number the initiator wishes to use to govern the association; and

`ros_data`: any initial data (this is a pointer to a `PElement` structure, which is fully explained in Chapter 5).

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

```
void    ROSFREE (ros)
struct RoSAPstart *ros;
```

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

If the call to `RoInit` is not successful, then a RO-P-REJECT.INDICATION event is simulated, and the relevant information is returned in an encoded `RoSAPindication` structure.

After examining the information returned by `RoInit` on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the association. For either response, the responder should use the `RoBeginResponse` routine (which corresponds to the RO-BEGIN.RESPONSE action).

```
int     RoBeginResponse (sd, status, data, roi)
int     sd;
int     status;
PE      data;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`status`: the acceptance indicator (either `ROS_ACCEPT` if the association is to be accepted, or one of the "fatal" user-initiated rejection codes listed in Table 3.1 on page 60);

`data`: any initial data if the association is to be accepted; and,

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

If the call to RoBeginResponse is successful, and the status parameter is set to ROS_ACCEPT, then association establishment has now been completed. If the call is successful, but the status parameter is not ROS_ACCEPT, then the association has been rejected and the responder may exit. Otherwise, if the call fails and the reason is "fatal", then the association is closed.

Client Initialization

A program wishing to associate itself with another user of remote operation services calls the RoBeginRequest routine, which corresponds to the RO-BEGIN.REQUEST action.

```
int     RoBeginRequest (called, data, roc, roi)
struct  RoSAPaddr *called;
PE      data;
struct RoSAPconnect *roc;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

called: the RoSAP address of the responder;

data: any initial data;

roc: a pointer to a RoSAPconnect structure, which is updated only if the call succeeds; and,

roi: a pointer to a RoSAPindication structure, which is updated only if the call fails.

If the call to RoBeginRequest is successful (this corresponds to a RO-BEGIN.CONFIRMATION event), it returns information in the roc parameter, which is a pointer to a RoSAPconnect structure.

```
struct RoSAPconnect {
    int     roc_sd;

    int     roc_result;

    PE      roc_data;
};
```

The elements of this structure are:

`roc_sd`: the association-descriptor to be used to reference this association;

`roc_result`: the association response; and,

`roc_data`: any initial data if the association was accepted.

If the call to `RoBeginRequest` is successful, and the `roc_result` element is set to `ROS_ACCEPT`, then association establishment has completed. If the call is successful, but the `roc_result` element is not `ROS_ACCEPT`, the the association attempt has been rejected, consult Table 3.1 to determine the reason for the reject. Otherwise, if the call fails then the association is not established and the `RoSAPpreject` structure of the `RoSAPindication` discriminated union has been updated.

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

```
void    ROCFREE (roc)
struct RoSAPconnect *roc;
```

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

Note that in the basic service the arguement parameter to an Invoke request cannot be NULL. As well only the initiator may make invoke requests.

A.1.3 Association Release

The `RoEndRequest` routine is used to request the release of an association, and corresponds to a RO-END.REQUEST action. This action may be taken by only the initiator of an association.

```
int     RoEndRequest (sd, priority, roi)
int     sd;
int     priority;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`priority:` the priority of this request; and,

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

If the call to `RoEndRequest` is successful, then the association has been released. Otherwise if the call fails and the error is not fatal, then the association remains established and the `RoSAPpreject` structure contained in the `RoSAPindication` parameter `roi` contains the reason for the failure.

Upon receiving a RO-END.INDICATION event, the user is required to generate a RO-END.RESPONSE action using the `RoEndResponse` routine.

```
int     RoEndResponse (sd, roi)
int     sd;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor; and,

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

If the call to `RoEndResponse` is successful, then the association has been released.

A.1.4 An Example

Let's consider how one might construct a generic server which uses remote operations services. This entity will use an asynchronous interface and the basic service discipline. Note that we could have selected the advanced service discipline by using `RtBInit` instead of `RoInit`, `RtBeginResponse` instead of `RoBeginResponse`, and `RtEndResponse` instead of `RoEndResponse` (respectively). This is demonstrated later in this appendix.

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 A.2, the initialization steps for the generic responder, including the outer *C* wrapper, is shown. First, the RoSAP state is re-captured by calling `RoInit`. 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 `RoBeginResponse` to accept the association. Then, `RoSetIndications` is set to specify an asynchronous event handler. Finally, the main request/reply loop is realized. The server simply uses *pause*(2) to wait for the next event.

Figure 3.2 on page 77 contains the definition of the `ros_indication` routine and associated routines. Since the remote operations service was used to establish the association, a different closing handler must be used. This is shown in Figure A.3.

A.2 Reliable Transfer

Under old-style association handling for reliable transfer, the X.410 OPEN and X.410 CLOSE primitives are used.

A.2.1 Addresses

Addresses at the reliable transfer service access point are represented by the `RtSAPaddr` structure.

```
struct RtSAPaddr {
    struct SSAPaddr rta_addr;

    u_short     rta_port;
};
```

This structure contains two elements:

`rta_addr`: the session address, as described in Section 3.2 on page 42 of *Volume Two*; and,

`rta_port`: the port number of the entity residing above the RtSAP.

In Figure A.4, an example of how one constructs the RtSAP address for the message transfer entity on host `RemoteHost` is presented. (The routine `is2saddr` is described on page 42 of *Volume Two*.)

```
#include <stdio.h>
#include "generic.h"            /* defines OPERATIONS and ERRORS */
#include <isode/rosap.h>
#include <isode/isoservent.h>


int     ros_indications ();


main (argc, argv, envp)                                                 10
int     argc;
char  **argv,
     **envp;
{
    int     result,
            sd;
    struct RoSAPstart   ross;
    register struct RoSAPstart *ros = &ross;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;                       20
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (RoInit (argc, argv, ros, roi) == NOTOK)
        error ("initialization fails:  %s", RoErrString (rop -> rop_reason));

    sd = ros -> ros_sd;
/*  do authentication using ros -> ros_data here... */
    ROSFREE (ros);

/* read command line arguments here... */                               30

/* could use ROS_VALIDATE or ROS_BUSY instead and then exit */

    if (RoBeginResponse (sd, ROS_ACCEPT, NULLPE, roi) == NOTOK)
        error ("RO-BEGIN.RESPONSE: %s", RoErrString (rop -> rop_reason));

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

    for (;;)                                                            40
        pause ();
}
```

Figure A.2: Initializing the generic ROS responder

```
...

static int  ros_end (sd, roe)
int     sd;
struct RoSAPend *roe;
{
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;
                                                                        10
    if (RoEndResponse (sd, roi) == NOTOK)
        error ("RO-END.RESPONSE: %s", RoErrString (rop -> rop_reason));

    exit (0);
}
```

Figure A.3: Finalizing the generic ROS responder

A.2.2 Association Establishment

The *librtsap*(3n) library distinguishes between the user which started an association, the *initiator*, and the user which was subsequently bound to the association, the *responder*. We sometimes term these two entities the *client* and the *server*, respectively.

Server Initialization

The *tsapd*(8c) daemon, upon accepting a connection from an initiating host, consults the ISO services database to determine which program on the local system implements the desired SSAP entity. In the case of the reliable transfer service, the *tsapd* program contains the bootstrap for the reliable transfer provider. The daemon will again consult the ISO services database to determine which program on the system implements the desired RtSAP entity.

Once the program has been ascertained, the daemon runs the program with any arguments listed in the database. In addition, it appends some *magic arguments* to the argument vector. Hence, the very first action performed by the responder is to re-capture the RtSAP state contained in the magic arguments. This is done by calling the routine `RtBInit`, which on a successful return, is equivalent to a X.410 OPEN.INDICATION from the reliable transfer service provider.

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

...

register struct SSAPaddr *sa;
struct RtSAPaddr rtas;
register struct RtSAPaddr *rta = &rtas;
register struct isoservent *is;

...

if ((is = getisoserventbyname ("p1", "rtsap")) == NULL)
    error ("rtsap/p1");

/* RemoteHost is the host we're interested in,
   e.g., "gremlin.nrtc.northrop.com" */

rta -> rta_port = is -> is_port;
if ((is = getisoserventbyname ("rts", "ssap")) == NULL)
    error ("ssap/rts");
if ((sa = is2saddr (RemoteHost, NULLCP, (struct isoservent *) 0))
        == NULLSA)
    error ("address translation failed");
rta -> rta_addr = *sa;

...
```

Figure A.4: Constructing the RtSAP address for the Message Transfer entity