Result Indication

When an RO-RESULT.INDICATION event occurs, the `roi_type` field of the `roi` parameter contains the value `ROI_RESULT`, and a `RoSAPresult` structure is contained inside the `roi` parameter.

```
struct RoSAPresult {
    int     ror_id;

    PE      ror_result;
};
```

The elements of this structure are:

`ror_id`: the invocation ID of this reply, which is identical to the ID of the request which generated the results;

`ror_op`: the operation code of the operation performed (meaningful only in the complete service discipline); and,

`ror_result`: the results of the operation.

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

```
void    RORFREE (ror)
struct RoSAPresult *ror;
```

The macro frees only the data allocated in the `RoSAPresult` structure and not the structure itself.

Under the basic service discipline, only the initiator to an association will receive this event in response to an earlier call to `RoInvokeRequest`.

Error Indication

When an RO-ERROR.INDICATION event occurs, the `roi_type` field of the `roi` parameter contains the value `ROI_ERROR`, and a `RoSAPerror` structure is contained inside the `roi` parameter.

```
struct RoSAPerror {
    int     roe_id;

    int     roe_error;
    PE      roe_param;
};
```

The elements of this structure are:

`roe_id`: the invocation ID of this reply, which is identical to the ID of the request which generated the error;

`roe_error`: the error code; and,

`roe_param`: the parameters of the error.

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

```
void    ROEFREE (roe)
struct RoSAPerror *roe;
```

The macro frees only the data allocated in the `RoSAPerror` structure and not the structure itself.

Under the basic service discipline, only the initiator to an association will receive this event in response to an earlier call to `RoInvokeRequest`.

User-Reject Indication

When an RO-U-REJECT.INDICATION event occurs, the `roi_type` field of the `roi` parameter contains the value `ROI_UREJECT`, and a `RoSAPureject` structure is contained inside the `roi` parameter.

```
struct RoSAPureject {
    int     rou_id;
    int     rou_noid;

    int     rou_reason;
};
```

The elements of this structure are:

`rou_id`: if `rou_noid` is not set, then this contains the invocation ID of the request which generated the rejection;

`rou_noid`: the invocation ID indicator (if set, then no request in particular caused the rejection to be generated); and,

`rou_reason`: the reason for the rejection (a "non-fatal" user-initiated rejection code listed in Table 3.1).

End Indication

When the `roi_type` field of the `roi` parameter contains the value `ROI_END`, a `RoSAPend` structure is contained inside the `roi` parameter.

```
struct RoSAPend {
    int     roe_dummy;
};
```

Depending on whether the reliable transfer service was used to start the association, a X.400 CLOSE.INDICATION or a RO-END.INDICATION event has occurred, and the user should respond appropriately.

Finish Indication

When the `roi_type` field of the `RoSAPindication` parameter `roi` contains the value `ROI_FINISH`, a `AcSAPfinish` structure is contained inside the `roi` parameter. Depending on whether the reliable transfer service was used to start the association, an RT-CLOSE.INDICATION or an A-RELEASE.INDICATION event has occurred, and the user should respond appropriately.

3.3.5 Rejecting Requests and Replies

The `RoURejectRequest` routine is used to perform user-level error-recovery. Usually, it signals the rejection of a previously received request or reply.

```
int     RoURejectRequest (sd, invoke, reason, priority, roi)
int     sd;
int    *invoke,
```

```
            reason,
            priority;
    struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`invoke:` a pointer to the invocation ID of the request in question (or `NULLIP` if this rejection does not apply to a particular request or reply);

`reason:` a "non-fatal" user-initiated rejection code as listed in Table 3.1 on page 60;

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

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

Upon a successful return from this call, a ROU-U-REJECT.INDICATION event has been queued for the other user.

3.3.6 Asynchronous Event Handling

The request/reply events discussed thus far have been synchronous in nature. Some users of the remote operations service may wish an asynchronous interface. The `RoSetIndications` routine is used to change the service associated with an association-descriptor to an asynchronous interface.

```
int   RoSetIndications (sd, indication, roi)
int   sd;
IFP   indication;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`indication:` the address of an event-handler routine to be invoked when an event occurs; and,

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

If the service is to be made asynchronous, then `indication` is specified; otherwise, if the service is to be made synchronous, it is not specified (use the manifest constant `NULLIFP`). The most likely reason for the call failing is `ROS_WAITING`, which indicates that an event is waiting for the user.

When an event occurs, the event-handler routine is invoked with two parameters:

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

The parameters are:

`sd`: the association-descriptor; and,

`roi`: a pointer to a `RoSAPindication` structure encoding the event.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released with the appropriate macro (i.e., `ROPFREE`, `ROXFREE`, `RORFREE`, or `ROEFREE`) when no longer needed.

When an event-handler is invoked, future invocations of the event-hander are blocked until it returns. The return value of the event-handler is ignored. Further, during the execution of a synchronous call to the library, the event-handler will be blocked from being invoked. The one exception to this is a call to `RoInvokeRequest` with the `class` parameter set to `ROS_SYNC`. In this circumstance, replies to invocations other than the one being waited for will result in the event-handler being invoked as appropriate.

> **NOTE:** The *librosap*(3n) library uses the SIGEMT signal to provide these services. Programs using asynchronous association-descriptors should NOT use SIGEMT for other purposes.

3.3.7 Synchronous Event Multiplexing

A user of the remote operations service may wish to manage multiple association-descriptors simultaneously; the routine `RoSelectMask` is provided for this

purpose. This routine updates a file-descriptor mask and associated counter for use with `xselect` (consult Section 2.4), as assocation-descriptors are file-descriptors.

```
int     RoSelectMask (sd, mask, nfds, roi)
int     sd;
fd_set *mask;
int    *nfds;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`mask`: a pointer to a file-descriptor mask meaningful to `xselect`;

`nfds`: a pointer to an integer-valued location meaningful to `xselect`; and,

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

If the call is successful, then the `mask` and `nfds` parameters can be used as arguments to `xselect`. The most likely reason for the call failing is `ROS_WAITING`, which indicates that an event is waiting for the user.

If `xselect` indicates that the association-descriptor is ready for reading, `RoWaitRequest` should be called with the `secs` parameter equal to `OK`. If the network activity does not constitute an entire event for the user, then `RoWaitRequest` will return `NOTOK` with error code `ROS_TIMER`.

3.4 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `roi` parameter given to the routine. The element called `roi_preject` in the `RoSAPindication` structure encodes the reason for the failure. To determine the reason, one coerces a pointer to a `RoSAPpreject` structure, and consults the `rop_reason` element of this latter structure. This element can be given as a parameter to the routine `RoErrString` which returns a null-terminated diagnostic string.

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

3.5 Compiling and Loading

Programs using the *librosap*(3n) library should include `<isode/rosap.h>`, which automatically includes the header file `<isode/psap.h>` described in Chapter 5 and the header file `<isode/acsap.h>` described in Chapter 2. The programs should also be loaded with `-lisode`.

3.6 Two Examples

Two examples are now presented: a detailed exposition on the construction of a responder for remote operations; and, a terse presentation of an initiator process.

3.6.1 The Generic Server

Let's consider how one might construct a generic server which uses remote operations services. This entity will use an asynchronous interface.

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 3.1, the initialization steps for the generic responder, including the outer *C* wrapper, is shown. First, the ACSE state is re-captured by calling `AcInit`. If this succeeds, then any command line arguments are parsed. Assuming that the responder is satisfied with the proposed association, it calls `AcAssocResponse` to accept the association. Then, `RoSetService` is called to set the underlying service to be used for remote operations. The `RoSetIndications` routine is called 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.

In Figure 3.2 on page 77, the `ros_indication` routine simply dispatches based on the value of the `roi_type` element of the `RoSAPindication` structure.

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


int     ros_indication ();


main (argc, argv, envp)
int     argc;                                                          10
char  **argv,
     **envp;
{
    int     result,
            sd;
    struct AcSAPstart   acss;
    register struct AcSAPstart *acs = &acss;
    struct AcSAPindication  acis;
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort   *aca = &aci -> aci_abort;             20
    register struct PSAPstart *ps = &acs -> acs_start;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (AcInit (argc, argv, acs, aci) == NOTOK)
        error ("initialization fails:  %s", AcErrString (aca -> aca_reason));

    sd = acs -> acs_sd;
    ACSFREE (acs);                                                     30

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

    if (AcAssocResponse (sd, ACS_ACCEPT, ACS_USER_NULL, NULLOID, NULLAEI,
                &ps -> ps_called, NULLPC, ps -> ps_defctxresult,
                ps -> ps_prequirements, ps -> ps_srequirements,
                SERIAL_NONE, ps -> ps_settings, &ps -> ps_connect,
                NULLPEP, 0, aci) == NOTOK)
        error ("A-ASSOCIATE.RESPONSE: %s", AcErrString (aca -> aca_reason));
                                                                       40
    if (RoSetService (sd, RoPService, roi) == NOTOK)
        error ("RoSetService:  %s", RoErrString (rop -> rop_reason));

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

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

Figure 3.1: Initializing the generic ROS responder

```
...

static int ros_indication (sd, roi)
int     sd;
register struct RoSAPindication *roi;
{
    switch (roi -> roi_type) {
        case ROI_INVOKE:
            ros_invoke (sd, &roi -> roi_invoke);
            break;                                                    10

        case ROI_RESULT:
            ros_result (sd, &roi -> roi_result);
            break;

        case ROI_ERROR:
            ros_error (sd, &roi -> roi_error);
            break;

        case ROI_UREJECT:                                             20
            ros_ureject (sd, &roi -> roi_ureject);
            break;

        case ROI_PREJECT:
            ros_preject (sd, &roi -> roi_preject);
            break;

        case ROI_FINISH:
            ros_finish (sd, &roi -> roi_finish);
            break;                                                    30

        default:
            error ("unknown indication type=%d", roi -> roi_type);
    }
}

...
```

Figure 3.2: The request/reply loop for the generic ROS responder

```
...

int     OP1 (), ..., OPn ();


/* OPERATIONS are numbered APDU_OPx, where each is a unique integer.  Further,
   APDU_UNKNOWN is used as a tag different than any valid operation.

   ERRORS are numbered ERROR_xyz, where each is a unique integer.
   ERROR_MISTYPED is used to signal an argument error to an operation.
   Further, ERROR_UNKNOWN is used as a tag to indicate that the operation
   succeeded.

   Finally, note that rox -> rox_args is updated in place by these routines.
   If the routine returns ERROR_UNKNOWN, then rox_args contains the results
   of the operation.  If the routine returns ERROR_MISTYPED, then rox_args is
   untouched.  Otherwise, if the routine returns any other value, then
   rox_args contains the parameters of the error which occurred.  Obviously,
   each routine calls ROXFREE prior to setting rox_args to a new value.
 */

static struct dispatch {
    int     ds_operation;
    IFP     ds_vector;
}       dispatches[] = {
    APDU_OP1,   OP1,

...

    APDU_OPn,   OPn,

    APDU_UNKNOWN
};

...
```

Figure 3.2: The request/reply loop for the generic ROS responder (continued)