```
        err,
        priority;
caddr_t out;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`id:` the ID of the invocation being responded to;

`err:` the error code being returned;

`out:` the address of the *C* structure for the error's parameter (if any);

`priority:` the priority of the response (use `ROS_NOPRIO` if the priority is undetermined); and,

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

The routine `RyDsUReject` is used to reject an invocation.

```
int RyDsUReject (sd, id, reason, priority, roi)
int     sd;
int id,
reason,
priority;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`id:` the ID of the invocation being rejected;

`reason:` the reason for the rejection (codes are listed in Table 3.1 on page 60 of *Volume One*);

`priority:` the priority of the response (use `ROS_NOPRIO` if the priority is undetermined); and,

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

8.7 Waiting for Events

The `RyWait` routine is used by both invokers and performers to wait for some event to occur. Normally, it is not used by invokers, except when it is desired to wait for a previously invoked asynchronous operation to complete.

```
int     RyWait (sd, id, out, secs, roi)
int     sd,
       *id,
        secs;
caddr_t *out;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`id:` the integer-value address of the invocation ID to wait for (use `NULLIP` if any invocation ID is satisfactory);

`out:` an address-valued location which is updated if the call succeeds;

`secs:` the maximum number of seconds to wait (a value of `NOTOK` indicates that the call should block indefinitely, whereas a value of `OK` indicates that the call should not block at all, e.g., a polling action); and,

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

The `RyWait` routine is analogous to the `RoWaitRequest` routine in the *librosy*(3n) library, with the additional functions of: automatically encoding and decoding arguments, results, and parameters; invoking a dispatch routine for the performer on incoming invocations; and, also queuing events for future retrieval.

The `RyWait` routine returns one of three values: `NOTOK` (on failure), `OK` (on reading a reply), or `DONE` (when something else happens).

If the call to `RyWait` returns the manifest constant `NOTOK`, then the `RoSAPpreject` structure contained in the `RoSAPindication` parameter `roi` contains the reason for the failure.

If the call to `RyWait` returns the manifest constant `OK`, then either a result or error has been returned from a previous invocation. The value of the `roi_type` element of the `RoSAPindication` parameter is either `ROI_RESULT` or `ROI_ERROR`, and the value of the address pointed to by `out` is a newly allocated *C* structure for the operations result or error's parameter (if any). Note that if a result or parameter is returned, then it is the invoker's responsibility to release the memory used by the *C* structure, when appropriate.

If the call to `RyWait` returns the manifest constant `DONE`, then an indication that the association is pending termination is returned. The value of the `roi_type` element of the `RoSAPindication` parameter is either `ROI_END` or `ROI_FINISH`. Consult Section 3.3.4 of *Volume One* for the details.

Note that because the *librosy*(3n) library may queue events while waiting for a particular event, it is important to call `RyWait` with a `secs` parameter of `OK` prior to deciding to explicitly block for activity (e.g., calling `RoSelectMask` or `xselect`).

8.8 Miscellaneous Routines

There are a few miscellaneous routines of interest.

8.8.1 Association Termination

When an association is either released or aborted, it is important to expunge any information regarding queued operations from the run-time environment. The `RyLose` routine is used to perform this function.

```
int     RyLose (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.

8.8.2 Utility Routines

It may be useful, primarily for diagnostic purposes, to retrieve an entry for a particular operation or error from the tables compiled by the *rosy*(1) compiler. There are four routines for this purpose.

The routine `findopbyop` takes a table and an operation code and returns the corresponding entry for the operation in the table, or the manifest constant `NULL` on failure.

```
struct RyOperation *findopbyop (ryo, op)
struct RyOperation *ryo;
int     op;
```

The parameters to this procedure are:

`ryo:` a pointer to a `RyOperation` table compiled by *rosy*; and,

`op:` the operation code of the operation to search for in the table.

The routine `findopbyname` takes a table and an operation name and returns the corresponding entry for the operation in the table, or the manifest constant `NULL` on failure.

```
struct RyOperation *findopbyname (ryo, name)
struct RyOperation *ryo;
char   *name;
```

The parameters to this procedure are:

`ryo:` a pointer to a `RyOperation` table compiled by *rosy*; and,

`name:` the operation name of the operation to search for in the table.

The routine `finderrbyerr` takes a table and an error code and returns the corresponding entry for the error in the table, or the manifest constant `NULL` on failure.

```
struct RyError *finderrbyerr (rye, err)
struct RyError *rye;
int     err;
```

The parameters to this procedure are:

`rye:` a pointer to a `RyError` table compiled by *rosy*; and,

`err:` the error code of the error to search for in the table.

The routine `finderrbyname` takes a table and an error code and returns the corresponding entry for the error in the table, or the manifest constant `NULL` on failure.

```
struct RyError *finderrbyname (rye, name)
struct RyError *rye;
char   *name;
```

The parameters to this procedure are:

`rye:` a pointer to a `RyError` table compiled by *rosy*; and,

`name:` the error name of the error to search for in the table.

8.9 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;
```

8.10 Compiling and Loading

Programs using the *librosy*(3n) library should include `<isode/rosy.h>`. The programs should also be loaded with `-lisode`.

Chapter 9

Boilerplate for Initiators

Let's consider how to build an initiator which is also an invoker. In Chapter 3, two forms for an initiator were identified: *interactive*, and *embedded*. The interactive initiator can be thought of as simply being a special case of an embedded initiator. Hence, we will start by describing the embedded initiator and then describe the additional structure added to form an interactive initiator.

If you have access to the source tree for this release, the directory *others/lookup/* contains the boilerplate described herein.

9.1 Embedded Initiator

An embedded initiator is characterized as automatically managing an association and invoking operations as required. There are four areas: association establishment, operation invocation, association release, and error handling.

9.1.1 Association Establishment

The application-entity information and presentation address for the desired service are computed, along with the application context and default presentation context information for the service. In addition, a session reference identifier is chosen.

The routine `AcAssocRequest` is called to establish an association for the service. The arguments are strictly boilerplate: they will work unaltered

for most applications. If the association is established, then the routine `RoSetService` is used to tell the remote operations library to use the presentation service as its underlying service.

...

```
#define myservice       "servicestore"
#define mycontext       "local service"
#define mypci           "local service pci"
```

...

```
    int         sd;
    struct SSAPref sfs;                                          10
    register struct SSAPref *sf;
    register struct PSAPaddr *pa;
    struct AcSAPconnect accs;
    register struct AcSAPconnect   *acc = &accs;
    struct AcSAPindication  acis;
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort *aca = &aci -> aci_abort;
    AEI         aei;
    OID         ctx,
                pci;                                             20
    struct PSAPctxlist pcs;
    register struct PSAPctxlist *pc = &pcs;
    struct RoSAPindication rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject *rop = &roi -> roi_preject;

    if ((aei = _str2aei (argv[1], myservice, mycontext,
                                isatty (fileno (stdin)), NULLCP, NULLCP))
            == NULLAEI)
        adios (NULLCP, "unable to resolve service:  %s", PY_pepy);   30
    if ((pa = aei2addr (aei)) == NULLPA)
        adios (NULLCP, "address translation failed");

    if ((ctx = ode2oid (mycontext)) == NULLOID)
        adios (NULLCP, "%s:  unknown object descriptor", mycontext);
```

```
    if ((ctx = oid_cpy (ctx)) == NULLOID)
        adios (NULLCP, "out of memory");
    if ((pci = ode2oid (mypci)) == NULLOID)
        adios (NULLCP, "%s:  unknown object descriptor", mypci);
    if ((pci = oid_cpy (pci)) == NULLOID)                                    40
        adios (NULLCP, "out of memory");
    pc -> pc_nctx = 1;
    pc -> pc_ctx[0].pc_id = 1;
    pc -> pc_ctx[0].pc_asn = pci;
    pc -> pc_ctx[0].pc_atn = NULLOID;

    if ((sf = addr2ref (PLocalHostName ())) == NULL) {
        sf = &sfs;
        (void) bzero ((char *) sf, sizeof *sf);
    }                                                                        50

    if (AcAssocRequest (ctx, NULLAEI, aei, NULLPA, pa, pc, NULLOID,
                        0, ROS_MYREQUIRE, SERIAL_NONE, 0, sf, NULLPEP, 0, NULLQOS,
                        acc, aci)
            == NOTOK)
        acs_adios (aca, "A-ASSOCIATE.REQUEST");

    if (acc -> acc_result != ACS_ACCEPT)
        adios (NULLCP, "association rejected:  [%s]",
                        AcErrString (acc -> acc_result));                    60

    sd = acc -> acc_sd;
    ACCFREE (acc);

    if (RoSetService (sd, RoPService, roi) == NOTOK)
        ros_adios (rop, "set RO/PS fails");

...
```

9.1.2 Operation Invocation

We'll consider how an operation is invoked using both the synchronous and asynchronous interfaces.

Synchronous Invocation

First, the argument (if any) for the operation is allocated and initialized. Then the macro `op_MODULE_operation` is called, which is really a call to the `RyOperation` routine described in Section 8.4 on page 108. One of three values is expected to be returned.

If the manifest constant `NOTOK` is returned, then some error has occured prior to invoking the operation. The most common error is the association being abruptly terminated due to network failure.

If the manifest constant `OK` is returned, then the responder replied with either a result or an error for the invocation. The variable `response` is consulted to determine if a result is present, or if not, which error is present. For each case, the `out` variable is cast to the appropriate variable, the application-specific code is executed, and then the structure is freed.

If the manifest constant `DONE` is returned, then the responder indicated that it wished to terminate the association. Since the association was established in a way which prohibited this behavior, this return is unlikely.

...

```
int         result,
            response;
struct type_MODULE_argument *arg;
caddr_t out;
struct RoSAPindication rois;
register struct RoSAPindication *roi = &rois;
register struct RoSAPpreject   *rop = &roi -> roi_preject;
                                                                    10
/* allocate and initialize "arg" here... */

switch (result = op_MODULE_operation (sd, arg, &out, &response, roi)) {
        case NOTOK:             /* failure */
            if (ROS_FATAL (rop -> rop_reason))
```

```
                ros_adios (rop, "OPERATION");
            ros_advise (rop, "OPERATION");
            break;

        case OK:                    /* got a result/error response */
            switch (response) {
                case RY_RESULT: /* got a result */
                    {
                        register struct type_MODULE_result *res =
                                (struct type_MODULE_result *) out;

/* do something with "res" here... */

                        free_MODULE_result (res);
                        break;
                    }

                case error_MODULE_error1:
                    {
                        register struct type_MODULE_parameter *parm =
                                (struct type_MODULE_parameter *) out;

/* do something with "parm" here... */

                        free_MODULE_parameter (parm);
                        break;
                    }

                /* other errors are handled here... */

                default:
                    adios (NULLCP, "unknown result/error returned (%d)",
                            response);
                    /* NOTREACHED */
            }
            break;

        case DONE:                  /* got RO-END? */
            adios (NULLCP, "got RO-END.INDICATION");
```