```
        return NOTOK;
    }
    advise (LLOG_NOTICE, NULLCP,
                "A-ASSOCIATE.INDICATION: <%d, %s, %s, %s, %d>",
                acs -> acs_sd, oid2ode (acs -> acs_context),
                sprintaei (&acs -> acs_callingtitle),
                sprintaei (&acs -> acs_calledtitle), acs -> acs_ninfo);
                                                                    130
    sd = acs -> acs_sd;

    for (vec++; *vec; vec++)
        advise (LLOG_EXCEPTIONS, NULLCP, "unknown argument \"%s\"", *vec);

    reply = startfnx ? (*startfnx) (sd, acs) : ACS_ACCEPT;

    result = AcAssocResponse (sd, reply, reply != ACS_ACCEPT
                                ? ACS_USER_NOREASON : ACS_USER_NULL,
                NULLOID, NULLAEI,                                   140
                NULLPA, NULLPC, ps -> ps_defctxresult,
                ps -> ps_prequirements, ps -> ps_srequirements,
                SERIAL_NONE, ps -> ps_settings, &ps -> ps_connect,
                NULLPEP, 0, aci);

    ACSFREE (acs);

    if (result == NOTOK) {
        acs_advise (aca, "A-ASSOCIATE.RESPONSE");
        return NOTOK;                                               150
    }
    if (reply != ACS_ACCEPT)
        return NOTOK;

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

    return sd;
}
                                                                    160
```

```
static int  ros_work (fd)
int         fd;
{
    int         result;
    caddr_t out;
    struct AcSAPindication  acis;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    switch (setjmp (toplevel)) {
        case OK:
            break;

        default:
            if (stopfnx)
                (*stopfnx) (fd, (struct AcSAPfinish *) 0);
        case DONE:
            (void) AcUAbortRequest (fd, NULLPEP, 0, &acis);
            (void) RyLose (fd, roi);
            return NOTOK;
    }

    switch (result = RyWait (fd, NULLIP, &out, OK, roi)) {
        case NOTOK:
            if (rop -> rop_reason == ROS_TIMER)
                break;
        case OK:
        case DONE:
            ros_indication (fd, roi);
            break;

        default:
            adios (NULLCP, "unknown return from RoWaitRequest=%d", result);
    }

    return OK;
}
```

```
static int ros_indication (sd, roi)
int       sd;
register struct RoSAPindication *roi;
{
   int       reply,
             result;

   switch (roi -> roi_type) {
         case ROI_INVOKE:                                          210
         case ROI_RESULT:
         case ROI_ERROR:
            adios (NULLCP, "unexpected indication type=%d", roi -> roi_type);
            break;

         case ROI_UREJECT:
            {
                  register struct RoSAPureject  *rou = &roi -> roi_ureject;

                  if (rou -> rou_noid)                             220
                     advise (LLOG_EXCEPTIONS, NULLCP,
                              "RO-REJECT-U.INDICATION/%d:  %s",
                              sd, RoErrString (rou -> rou_reason));
                  else
                     advise (LLOG_EXCEPTIONS, NULLCP,
                              "RO-REJECT-U.INDICATION/%d:  %s (id=%d)",
                              sd, RoErrString (rou -> rou_reason),
                              rou -> rou_id);
            }
            break;                                                230

         case ROI_PREJECT:
            {
                  register struct RoSAPpreject  *rop = &roi -> roi_preject;

                  if (ROS_FATAL (rop -> rop_reason))
                     ros_adios (rop, "RO-REJECT-P.INDICATION");
                  ros_advise (rop, "RO-REJECT-P.INDICATION");
            }
```

```
            break;                                                  240

        case ROI_FINISH:
            {
                register struct AcSAPfinish *acf = &roi -> roi_finish;
                struct AcSAPindication  acis;
                register struct AcSAPabort *aca = &acis.aci_abort;

                advise (LLOG_NOTICE, NULLCP, "A-RELEASE.INDICATION/%d:  %d",
                        sd, acf -> acf_reason);
                                                                    250
                reply = stopfnx ? (*stopfnx) (sd, acf) : ACS_ACCEPT;

                result = AcRelResponse (sd, reply, ACR_NORMAL, NULLPEP,
                            0, &acis);

                ACFFREE (acf);

                if (result == NOTOK)
                    acs_advise (aca, "A-RELEASE.RESPONSE");
                else                                                260
                    if (reply != ACS_ACCEPT)
                        break;
                longjmp (toplevel, DONE);
            }
        /* NOTREACHED */

        default:
            adios (NULLCP, "unknown indication type=%d", roi -> roi_type);
    }
}                                                                   270


static int  ros_lose (td)
struct TSAPdisconnect *td;
{
    if (td -> td_cc > 0)
        adios (NULLCP, "TNetAccept:  [%s] %*.*s",
                TErrString (td -> td_reason), td -> td_cc, td -> td_cc,
```

```
                td -> td_data);
    else                                                                    280
        adios (NULLCP, "TNetAccept:  [%s]", TErrString (td -> td_reason));
}

...
```

10.1.2 Operation Response

When an operation is invoked, its dispatch routine is called from the routine `RyWait` as described in Section 8.6.1 on Page 114. The boilerplate for a dispatch routine is fairly uniform. A check is made to see if the invocation is linked to a previous invocation, and the invocation is logged. Any user-rejections are performed by the routine `ureject` which is a simple wrapper for the `RyDsUReject` routine. Next, the operation is attempted. If it succeeded (as denoted by `won`), then a result is allocated, and initialized and returned to the initiator by the `RyDsResult` routine. Otherwise, an error is selected, and its parameter is allocated, initialized and returned to the initiator by the `error` routine which is a simple wrapper for the `RyDsError` routine. Finally, the argument is freed and the handler returns.

```
...

static int  op_name (sd, ryo, rox, in, roi)
int         sd;
struct RyOperation *ryo;
struct RoSAPinvoke *rox;
caddr_t   in;
struct RoSAPindication *roi;
{
    int         result,                                          10
                won;
    register struct type_MODULE_argument *arg =
                    (struct type_MODULE_argument *) in;

/* this check should *not* be made if linking of operations is expected */
    if (rox -> rox_nolinked == 0) {
            advise (LLOG_EXCEPTIONS, NULLCP,
                    "RO-INVOKE.INDICATION/%d:  %s, unknown linkage %d",
                    sd, ryo -> ryo_name, rox -> rox_linkid);
            return ureject (sd, ROS_IP_LINKED, rox, roi);          20
    }
    if (debug)
            advise (LLOG_DEBUG, NULLCP, "RO-INVOKE.INDICATION/%d:  %s",
                    sd, ryo -> ryo_name);
```

```
/* do something with "arg" here and decide if won... */

    if (won) {
        register struct type_MODULE_result *res;
                                                                    30
/* allocate and initialize the result "res" here... */

        if (RyDsResult (sd, rox -> rox_id, (caddr_t) res, ROS_NOPRIO, roi)
                == NOTOK)
            ros_adios (&roi -> roi_preject, "RESULT");
        result = OK;

        free_MODULE_result (res);
    }
    else {                                                          40
        struct type_MODULE_parameter *parm;

/* allocate and initialize "parm" here... */

        result = error (sd, error_MODULE_error1, (caddr_t) parm, rox, roi);

        free_MODULE_parameter (parm);
    }

/* "arg" will be automatically free'd when op_name returns */       50

    return result;
}

...

static int  error (sd, err, param, rox, roi)
int         sd,
            err;
caddr_t     param;                                                  60
struct RoSAPinvoke *rox;
struct RoSAPindication *roi;
{
    if (RyDsError (sd, rox -> rox_id, err, param, ROS_NOPRIO, roi) == NOTOK)
```

```
        ros_adios (&roi -> roi_preject, "ERROR");

    return OK;
}

                                                                    70
static int  ureject (sd, reason, rox, roi)
int         sd,
            reason;
struct RoSAPinvoke *rox;
struct RoSAPindication *roi;
{
    if (RyDsUReject (sd, rox -> rox_id, reason, ROS_NOPRIO, roi) == NOTOK)
        ros_adios (&roi -> roi_preject, "U-REJECT");

    return OK;                                                      80
}

...
```

10.1.3 Error Handling

These routines for the most part are all straight-forward.

`ros_adios:` used to report a ROS error and terminate;

`ros_advise:` used to report a ROS error;

`acs_advise:` used to report an ACS error;

`adios:` used to report an error and terminate;

`advise:` used to report an error; and,

pepy generate errors are normally caught by the routine `PY_advise` by using the `-a` switch to *pepy*. The *librosy*(3n) routines use these to build up error message when *pepy* routines fail.

...

```
void    ros_adios (rop, event)
register struct RoSAPpreject *rop;
char   *event;
{
    ros_advise (rop, event);

    longjmp (toplevel, NOTOK);
}
```

```
void    ros_advise (rop, event)
register struct RoSAPpreject *rop;
char   *event;
{
    char    buffer[BUFSIZ];

    if (rop -> rop_cc > 0)
        (void) sprintf (buffer, "[%s] %*.*s", RoErrString (rop -> rop_reason),
                rop -> rop_cc, rop -> rop_cc, rop -> rop_data);
    else
        (void) sprintf (buffer, "[%s]", RoErrString (rop -> rop_reason));
```

```
    advise (LLOG_EXCEPTIONS, NULLCP, "%s:  %s", event, buffer);
}


void    acs_advise (aca, event)
register struct AcSAPabort *aca;
char   *event;
{
    char    buffer[BUFSIZ];

    if (aca -> aca_cc > 0)
        (void) sprintf (buffer, "[%s] %*.*s",
                AcErrString (aca -> aca_reason),
                aca -> aca_cc, aca -> aca_cc, aca -> aca_data);
    else
        (void) sprintf (buffer, "[%s]", AcErrString (aca -> aca_reason));

    advise (LLOG_EXCEPTIONS, NULLCP, "%s:  %s (source %d)", event, buffer,
                aca -> aca_source);
}


#ifndef lint
void    adios (va_alist)
va_dcl
{
    va_list ap;

    va_start (ap);

    _ll_log (pgm_log, LLOG_FATAL, ap);

    va_end (ap);

    _exit (1);
}
#else
/* VARARGS2 */
```