```
            /* NOTREACHED */

        default:
            adios (NULLCP, "unknown return from RyOperation=%d", result);
            /* NOTREACHED */
    }                                                                    60

    free_MODULE_argument (arg);

...
```

ASynchronous Invocation

```
...

int     do_result (), do_error ();

...

    int     result;
    struct RoSAPindication rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

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

    switch (result = stub_MODULE_operation (sd, RyGenID (sd), arg, do_result,
                                            do_error, ROS_SYNC, roi)) {
        case NOTOK:             /* failure */
            if (ROS_FATAL (rop -> rop_reason))
                ros_adios (rop, "STUB");
            ros_advise (rop, "STUB");
            break;

        case OK:
            break;

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

        default:
            adios (NULLCP, "unknown return from RyStub=%d", result);
            /* NOTREACHED */
    }

    free_MODULE_argument (arg);

...
```

```
do_result (sd, id, dummy, result, roi)
int       sd;
int       id,                                                    40
          dummy;
struct type_MODULE_result *result;
struct RoSAPindication *roi;
{
   /* do something with "result" here... */

   /* "result" will be automatically free'd when do_result returns */

   return OK;
}                                                                 50

...


do_error (sd, id, error, parameter, roi)
int       sd;
int       id,
          error;
caddr_t parameter;
struct RoSAPindication *roi;                                      60
{
   switch (error) {
          case error_MODULE_error1:
          {
             register struct type_MODULE_parameter *parm =
                              (struct type_MODULE_parameter *) parameter;

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

             break;                                               70
          }

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


          default:
```

```
            adios (NULLCP, "unknown error returned (%d)", error);
            /* NOTREACHED */
    }
                                                                    80
    /* "parameter" will be automatically free'd when do_error returns */

    return OK;
}

...
```

9.1.3 Association Release

Terminating the assocation is simple: the routine `AcRelRequest` is called.

```
...

    struct AcSAPrelease acrs;
    register struct AcSAPrelease   *acr = &acrs;
    struct AcSAPindication  acis;
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort *aca = &aci -> aci_abort;

    if (AcRelRequest (sd, ACF_NORMAL, NULLPEP, 0, NOTOK, acr, aci) == NOTOK)
            acs_adios (aca, "A-RELEASE.REQUEST");                          10

    if (!acr -> acr_affirmative) {
            (void) AcUAbortRequest (sd, NULLPEP, 0, aci);
            adios (NULLCP, "release rejected by peer:  %d", acr -> acr_reason);
    }

    ACRFREE (acr);

...
```

9.1.4 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_adios:` used to report an ACS error and terminate;

`acs_advise:` used to report an ACS error;

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

`advise:` used to report an error.

It is assumed that there is a definition of the form

```
char   *myname = "myname";
```

for use by the `advise` routine.

...

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

   _exit (1);
}
```

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


void    acs_adios (aca, event)
register struct AcSAPabort *aca;                                    30
char   *event;
{
    acs_advise (aca, event);

    _exit (1);
}


void    acs_advise (aca, event)
register struct AcSAPabort *aca;                                    40
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));   50

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


#ifndef lint
void    _advise ();
```

```
                                                                    60
void    adios (va_alist)
va_dcl
{
    va_list ap;

    va_start (ap);

    _advise (ap);

    va_end (ap);                                                    70

    _exit (1);
}
#else
/* VARARGS */

void    adios (what, fmt)
char   *what,
       *fmt;
{                                                                   80
    adios (what, fmt);
}
#endif


#ifndef lint
void    advise (va_alist)
va_dcl
{
    va_list ap;                                                     90

    va_start (ap);

    _advise (ap);

    va_end (ap);
}
```

```
static void _advise (ap)                                        100
va_list    ap;
{
    char    buffer[BUFSIZ];

    asprintf (buffer, ap);

    (void) fflush (stdout);

    fprintf (stderr, "%s:    ", myname);
    (void) fputs (buffer, stderr);                              110
    (void) fputc ('\n', stderr);

    (void) fflush (stderr);
}
#else
/* VARARGS */

void      advise (what, fmt)
char   *what,
       *fmt;                                                    120
{
    advise (what, fmt);
}
#endif
```

9.2 Interactive Initiator

Now, let's build on these routines to write an initiator which is interactive: the user runs a program and interactively directs the invocation of operations.

9.2.1 Include File

Let's consider what an `#include` file, say `ryinitiator.h`, might look like. First, the standard *librosy*(3n) definitions are included, then a `dispatch` structure is defined, along with the boilerplate routines. The `dispatch` structure will be used by the boilerplate to invoke a user-supplied routine that will invoke the operation.

```
/* ryinitiator.h -- include file for the generic interactive initiator */

#include <isode/rosy.h>


static struct dispatch {
    char    *ds_name;
    int     ds_operation;

    IFP     ds_argument;
    IFP     ds_free;

    IFP     ds_result;
    IFP     ds_error;

    char    *ds_help;
};


void    adios (), advise ();
void    acs_adios (), acs_advise ();
void    ros_adios (), ros_advise ();

int     ryinitiator ();
```