```
    asprintf (buffer, ap);                                          340

    (void) fflush (stdout);

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

    (void) fflush (stderr);
}
#else                                                               350
/* VARARGS */

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

9.2.3 An Example

An example of an interactive initiator written using this boilerplate is shown in Section 11.3 on page 178. However, a brief exposition of the `dispatch` structure and possible `help` and `quit` routines are shown here.

```
...

static struct dispatch dispatches[] = {
/* real user-supplied routines go here... */

    "help", 0, do_help, NULLIFP, NULLIFP, NULLIFP,
        "print this information",
    "quit", 0, do_quit, NULLIFP, NULLIFP, NULLIFP,
        "terminate the association and exit",
                                                        10
    NULL
};


/* ARGSUSED */

static int  do_help (sd, ds, args, dummy)
int         sd;
register struct dispatch *ds;
char  **args;                                           20
caddr_t *dummy;
{
    printf ("\nCommands are:\n");
    for (ds = dispatches; ds -> ds_name; ds++)
        printf ("%s\t%s\n", ds -> ds_name, ds -> ds_help);

    return NOTOK;
}


                                                        30
/* ARGSUSED */

static int  do_quit (sd, ds, args, dummy)
```

```
int       sd;
struct dispatch *ds;
char  **args;
caddr_t *dummy;
{
    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");

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

    ACRFREE (acr);

    exit (0);
}
```

Chapter 10

Boilerplate for Responders

Let's consider how to build a responder which is also a performer. In Chapter 3, two forms for a responder were identified: *dynamic*, in which each incoming association caused the instantiation of a new responder, and, *static*, in which each incoming association was given to a pre-existing single instantiation of a responder. The dynamic responder can be thought of as simply being a special case of a static responder. Hence, we will describe how one builds a static responder in this chapter.

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

10.1 Static Responder

There are three areas for the boilerplate: association management, operation response, and, error handling.

Before proceeding however, let's consider what an `#include` file, say `ryresponder.h`, might look like. First, the standard *librosy*(3n) definitions are included, along with the definitions for the daemon logging package. Next, 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 respond to an invocation.

```
/* ryresponder.h - include file for the generic idempotent responder */

#include <isode/rosy.h>
#include <isode/logger.h>


static struct dispatch {
    char    *ds_name;
    int     ds_operation;

    IFP     ds_vector;
};


extern int  debug;


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

int     ryresponder ();
```

10.1.1 Association Management

Association management is performed precisely the way it is outlined in Section 2.5 of *Volume One*. This is implemented by the routine `ryresponder`:

```
int     ryresponder (argc, argv, myservice, dispatches, ops,
                start, stop)
int     argc;
char  **argv,
       *myservice;
struct dispatch *dispatches;
struct RyOperation *ops;
IFP     start,
        stop;
```

The parameters to this procedure are:

`argc/argv:` the argument vector (and its length) that the program was invoked with;

`myservice:` the non-host portion of the application-entity information;

`dispatches:` a pointer to a `dispatch` table;

`ops:` a pointer to a `RyOperation` table;

`start:` the address of a routine to decide if incoming associations should be accepted (use `NULLIFP` if associations should always be accepted); and,

`stop:` the address of a routine to note that an association requests termination or has been terminated (use `NULLIFP` if associations termination is unremarkable).

The function of this routine is straight-forward though tedious. First, `myname` is initialized to the name that the program was invoked with. Next, a debug flag is possibly set and the daemon logging package is initialized, and the responder's application-entity information is computed. Finally, each operation is registered with the `RyDispatch` routine, and the `start` and `stop` routines are remembered.

The routine `isodeserver` is then called to manage any associations. As a result, the `ros_init` routine will be informed of new associations, the `ros_work` routine will be informed of network activity, and the `ros_lose` routine will be advised if network listening fails.

The `ros_init` routine is also straight-forward. First, `AcInit` is called to recapture the ACSE-state, then the user-defined association acceptance routine is called (if any). This routine should return either `ACS_ACCEPT` if the association is to be accepted, or either

```
ACS_TRANSIENT, ACS_PERMANENT,
```

otherwise (these latter two codes are discussed in Table 2.1 on page 29 of *Volume One*). The routine `AcAssocResponse` is then called to deal with the incoming association. The arguments are strictly boilerplate: they will work

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

The routine `ros_work` is called when activity occurs on an association, and is somewhat complex. The routine sets a global return vector using *setjmp* (3) and then calls `RyWait` to poll for the next operation-related event. This usually results in one of the operations registered earlier being dispatched immediately, and then `RyWait` will return `NOTOK` with an error condition of `ROS_TIMER` indicating that there is no more network activity pending. Otherwise, the routine `ros_indication` is called to handle extraordinary conditions on the association. If some error occurred during the handling of an invocation, use of the routine `ros_adios` will cause control to return to the `setjmp` call. In this case, several things happen. First, the user-defined association termination routine routine is called (if any). This routine should note that the association is now abruptly terminated. Next, the `AcUAbortRequest` routine is called to make sure that the association is (ungracefully) released. Following this, the `RyLose` routine is called to expunge any information regarding queued operations from the run-time environment. Finally, `NOTOK` is returned to `isodeserver`, which causes the association to be removed from the list of current associations.

The `ros_indication` routine is used to handle uncommon events for an association: user-rejections, provider-rejections, and association termination. In all three cases, the event is logged. In the case of the initiator requesting that the association be terminated, several things happen. First, the user-defined association termination routine is called (if any). This routine should return either `ACS_ACCEPT` if the association is to be released, or `ACS_REJECT` if the termination is to be refused. The routine `AcRelResponse` is then called to deal with the request to terminate the association. If the termination was accepted, then control returns to the `setjmp` call in `ros_work`, which finalizes things.

The `ros_lose` routine is simple: the error condition is logged.

/ ryresponder.c — generic idempotent responder */*

```
#include <stdio.h>
#include <setjmp.h>
#include "ryresponder.h"
```

```
#include <isode/tsap.h>         /* for listening */

/* DATA */

int     debug = 0;                                                      10

static LLog _pgm_log = {
    "responder.log", NULLCP, NULLCP,
    LLOG_FATAL | LLOG_EXCEPTIONS | LLOG_NOTICE, LLOG_FATAL, -1,
    LLOGCLS | LLOGCRT | LLOGZER, NOTOK
};
LLog *pgm_log = &_pgm_log;

static char *myname = "ryresponder";
                                                                        20

static jmp_buf toplevel;


static IFP      startfnx;
static IFP      stopfnx;

int     ros_init (), ros_work (), ros_indication (), ros_lose ();


                                                                        30
extern int  errno;

/* RESPONDER */

int         ryresponder (argc, argv, host, myservice, mycontext,
                dispatches, ops, start, stop)
int     argc;
char  **argv,
      *host,
      *myservice,                                                       40
      *mycontext;
struct dispatch *dispatches;
struct RyOperation *ops;
IFP     start,
```

```
        stop;
{
    register struct dispatch   *ds;
    AEI     aei;
    struct TSAPdisconnect   tds;
    struct TSAPdisconnect  *td = &tds;                                  50
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (myname = rindex (argv[0], '/'))
        myname++;
    if (myname == NULL || *myname == NULL)
        myname = argv[0];

    isodetailor (myname, 0);                                            60
    if (debug = isatty (fileno (stderr)))
        ll_dbinit (pgm_log, myname);
    else {
        static char  myfile[BUFSIZ];

        (void) sprintf (myfile, "%s.log",
                        (strncmp (myname, "ros.", 4)
                                && strncmp (myname, "lpp.", 4))
                            || myname[4] == NULL
                        ? myname : myname + 4);                         70
        pgm_log -> ll_file = myfile;
        ll_hdinit (pgm_log, myname);
    }

    advise (LLOG_NOTICE, NULLCP, "starting");

    if ((aei = _str2aei (host, myservice, mycontext, 0, NULLCP, NULLCP))
            == NULLAEI)
        adios (NULLCP, "unable to resolve service:  %s", PY_pepy);
                                                                        80
    for (ds = dispatches; ds -> ds_name; ds++)
        if (RyDispatch (NOTOK, ops, ds -> ds_operation, ds -> ds_vector, roi)
                == NOTOK)
```

```
            ros_adios (rop, ds -> ds_name);

    startfnx = start;
    stopfnx = stop;

    if (isodeserver (argc, argv, aei, ros_init, ros_work, ros_lose, td)
            == NOTOK) {                                                    90
        if (td -> td_cc > 0)
            adios (NULLCP, "isodeserver:  [%s] %*.*s",
                    TErrString (td -> td_reason),
                    td -> td_cc, td -> td_cc, td -> td_data);
        else
            adios (NULLCP, "isodeserver:  [%s]",
                    TErrString (td -> td_reason));
    }

    return 0;                                                              100
}


static int  ros_init (vecp, vec)
int         vecp;
char  **vec;
{
    int         reply,
                result,
                sd;                                                        110
    struct AcSAPstart   acss;
    register struct AcSAPstart *acs = &acss;
    struct AcSAPindication  acis;
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort   *aca = &aci -> aci_abort;
    register struct PSAPstart *ps = &acs -> acs_start;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;
                                                                           120
    if (AcInit (vecp, vec, acs, aci) == NOTOK) {
        acs_advise (aca, "initialization fails");
```