vgrindefs, 36, 76

W
Walton, Simon, xvi
Weller, Daniel, xviii
Wenzel, Oliver, xviii
Wilder, Rick, xviii
Willson, Stephen H., xv
Worsley, Andrew, xvii

Y
yacc, 27, 37, 57

libtsap, xviii, 8
lint, 10, 34, 35

M
make, 5, 23, 35, 51, 52, 75, 88
McLoughlin, L., xviii
Michaelson, George, xv
Miller, Steve D., xvii
Moore, Christopher W., xv

N
NBS, xvii
NIST, xvii
Nordmark, Erik, xviii

O
Onions, Julian, xvi

P
Pavel, John, xvi
Pavlou, George, xviii
pepsy, xvii, 10, 35, 86, 87, 88
pepy, xvi, 9, 10, 22, 34, 37, 46, 50, 57, 58, 60, 61, 65, 70, 71, 72, 73, 74, 75, 76, 86, 87, 88, 151
perror, 72
posy, xvii, 9, 10, 22, 35, 37, 41, 44, 46, 50, 51, 57, 86, 87, 88
Prafullchandra, Hemma, xviii
Preuss, Don, xviii
Pring, Ed, xix
printf, 72
PY_advise, 72, 151
PY_pepy, 73

Q
quipu, 9

R
Reinart, John A., xviii
Rekhter, Jacob, xviii
Robbins, Colin J., xvi
Roe, Mike, xvi
RoErrString, 107
Romine, John L., xv
ros.image, 21
rosy, 9, 21, 27, 33, 34, 35, 37, 45, 73, 86, 91, 92, 105, 106, 107
Ruttle, Keith, xvi
RyDiscard, 99
RyDispatch, 100
RyDsError, 102
RyDsResult, 102
RyDsUReject, 103
RyError, 93
ryinitiator, 123
RyLose, 105
RyOperation, 92, 94
RyOpInvoke, 95
ryresponder, 137
RyStub, 97
RyWait, 104

S
Scott, John A., xvii
setjmp, 139
Srinivasan, Raj, xviii

T
tar, xiii
Taylor, Jem, xvii
testdebug, 73
tgrind, 36, 76
Titcombe, Steve, xvi
Turland, Alan, xvi

U
U.C. Berkeley, xix

V
Vanderbilt, Peter, xviii
vgrind, 36, 76

Index

A
aetbuild, 10
Aziz, Ashar, xviii

B
Braun, Hans-Werner, xvii
Brezak, John, xviii

C
Cass, Dwight E., xv
Chirieleison, Don, xviii
Cowin, Godfrey, xvi
cpp, 75

D
dish, 9
dsabuild, 10
Dubous, Olivier, xvii

E
Easterbrook, Stephen, xvi
EEC, xvi
ESPRIT, xvi

F
finderrbyerr, 106
finderrbyname, 107
findopbyname, 106
findopbyop, 106
Finni, Olli, xvii

G
Gosling, James, xix

H
Heinänen, Juha, xviii
Horton, Mark R., xvii
Horvath, Nandor, xviii

I
isobjects, 20, 155
isoentities, 21
isoservices, 21

J
Jacobsen, Ole-Jorgen, xviii
Jelfs, Philip B., xviii
Jordan, Kevin E., xviii

K
Keogh, Paul, xviii
Kille, Stephen E., xvi
Knight, Graham, xviii

L
Lamport, Leslie, xix
LaTeX, xix, 178
Lavender, Greg, xvii
libacsap, 7, 94, 100
libdsap, 9
libpepy, 73
libpsap, xvi, 7, 9
libpsap2, 8
libpsap2-lpp, 8
librosap, xv, 7, 91, 96
librosy, 9, 27, 34, 35, 91, 92, 99, 101, 104, 105, 107, 122, 136, 151
librtsap, 7
libssap, 8

[ISO88a] Information Processing Systems — Text Communication — Remote Operations Part 1: Model, Notation and Service Definition. International Organization for Standardization and International Electrotechnical Committee, March, 1988. Working Document for International Standard 9072-1.

[ISO88b] Information Processing Systems — Text Communication — Remote Operations Part 2: Protocol Specification. International Organization for Standardization and International Electrotechnical Committee, March, 1988. Working Document for International Standard 9072-2.

[JPost81] Jon B. Postel. *Transmission Control Protocol.* Request for Comments 793, DDN Network Information Center, SRI International, September, 1981. See also MIL-STD 1778.

[MRose86] Marshall T. Rose and Dwight E. Cass. OSI Transport Services on top of the TCP. *Computer Networks and ISDN Systems*, 12(3), 1986. Also available as NRTC Technical Paper #700.

[MRose90] Marshall T. Rose. *The Open Book: A Practical Perspective on Open Systems Interconnection.* Prentice-hall, 1990. ISBN 0–13–643016–3.

Bibliography

[BKern78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language. Software Series*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[CCITT88a] Remote Operations: Model, Notation and Service Definition. International Telegraph and Telephone Consultative Committee, March, 1988. Recommendation X.219.

[CCITT88b] Remote Operations: Protocol Specification. International Telegraph and Telephone Consultative Committee, March, 1988. Recommendation X.229.

[CCITT88c] Specification of Abstract Syntax Notation One. International Telegraph and Telephone Consultative Committee, 1988. Recommendation X.208.

[ECMA85] Remote Operations: Concepts, Notation and Connection-Oriented Mappings. December, 1985. ECMA TR/31.

[ECoop82] Eric C. Cooper. *Writing Distributed Programs with Courier.* Technical Report, Computer Science Division — EECS, University of California, Berkeley, March, 1982.

[ISO87] Information Processing — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1). International Organization for Standardization and International Electrotechnical Committee, 1987. International Standard 8824.

```
int       sd,
          reason;                                              190
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;
}
```

Figure 11.4: Responder for the Password Lookup Service

```
            res -> uid -> arg = pw -> pw_uid;                                  150
            res -> gid -> arg = pw -> pw_gid;
            res -> quota = pw -> pw_quota;

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

        free_PasswordLookup_Passwd (res);                                      160
    }
    else
        result = error (sd, error_PasswordLookup_noSuchUser, (caddr_t) NULL,
                rox, roi);

    return result;
}


/* ERROR */                                                                    170

static int  error (sd, err, param, rox, roi)
int         sd,
            err;
caddr_t     param;
struct RoSAPinvoke *rox;
struct RoSAPindication *roi;
{
    if (RyDsError (sd, rox -> rox_id, err, param, ROS_NOPRIO, roi) == NOTOK)
        ros_adios (&roi -> roi_preject, "ERROR");                              180

    return OK;
}


/* U-REJECT */

static int  ureject (sd, reason, rox, roi)
```

```
        advise (LLOG_DEBUG, NULLCP, "RO-INVOKE.INDICATION/%d:  %s",
                sd, ryo -> ryo_name);

    return lookup (sd, getpwuid (arg -> arg), rox, roi);
}


static int  lookup (sd, pw, rox, roi)
int     sd;
struct passwd *pw;                                              120
struct RoSAPinvoke *rox;
struct RoSAPindication *roi;
{
    int     result;

    if (pw) {
        register struct type_PasswordLookup_Passwd *res = NULL;

        if (xalloc (res, struct type_PasswordLookup_Passwd *) == NULL
                || (res -> name = salloc (pw -> pw_name)) == NULL       130
                || (*pw -> pw_passwd
                        && (res -> passwd = salloc (pw -> pw_passwd)) == NULL)
                || xalloc (res -> uid, struct type_PasswordLookup_UserID *)
                        == NULL
                || xalloc (res -> gid, struct type_PasswordLookup_GroupID *)
                        == NULL
                || (pw -> pw_comment
                        && (res -> comment = salloc (pw -> pw_comment))
                                == NULL)
                || (pw -> pw_gecos                                      140
                        && (res -> gecos = salloc (pw -> pw_gecos)) == NULL)
                || (pw -> pw_dir
                        && (res -> dir = salloc (pw -> pw_dir)) == NULL)
                || (pw -> pw_shell
                        && (res -> shell = salloc (pw -> pw_shell))
                                == NULL))
            result = error (sd, error_PasswordLookup_congested, (caddr_t) NULL,
                rox, roi);
        else {
```

```
    if (rox -> rox_nolinked == 0) {
        advise (LLOG_NOTICE, NULLCP,
                "RO-INVOKE.INDICATION/%d:  %s, unknown linkage %d",
                sd, ryo -> ryo_name, rox -> rox_linkid);
        return ureject (sd, ROS_IP_LINKED, rox, roi);
    }
    if (debug)
        advise (LLOG_DEBUG, NULLCP, "RO-INVOKE.INDICATION/%d:  %s",
                sd, ryo -> ryo_name);

    if ((cp = qb2str (arg)) == NULL)
        result = error (sd, error_PasswordLookup_congested, (caddr_t) NULL,
                        rox, roi);
    else {
        result = lookup (sd, getpwnam (cp), rox, roi);
        free (cp);
    }

    return result;
}


static int  op_lookupUID (sd, ryo, rox, in, roi)
int         sd;
struct RyOperation *ryo;
struct RoSAPinvoke *rox;
caddr_t  in;
struct RoSAPindication *roi;
{
    register struct type_PasswordLookup_UserID   *arg =
                (struct type_PasswordLookup_UserID   *) in;

    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);
    }
    if (debug)
```

```
    "lookupUser", operation_PasswordLookup_lookupUser, op_lookupUser,

    "lookupUID", operation_PasswordLookup_lookupUID, op_lookupUID,

    NULL
};


/* MAIN */

/* ARGSUSED */

main (argc, argv, envp)
int         argc;
char  **argv,
      **envp;
{
    ryresponder (argc, argv, PLocalHostName (), myservice, mycontext,
                    dispatches, table_PasswordLookup_Operations,
                    NULLIFP, NULLIFP);

    exit (0);                           /* NOTREACHED */
}


/* OPERATIONS */

static int  op_lookupUser (sd, ryo, rox, in, roi)
int         sd;
struct RyOperation *ryo;
struct RoSAPinvoke *rox;
caddr_t  in;
struct RoSAPindication *roi;
{
    int     result;
    char   *cp;
    register struct type_PasswordLookup_UserName   *arg =
                    (struct type_PasswordLookup_UserName   *) in;
```

11.4 Responder

The responder for the service is placed in *lookupd.c*. This file is shown in Figure 11.4 starting on page 175. Note that if actual network addresses were associated with the service definitions made earlier, then the responder could operate using the static discipline.

```
/* lookup_service.c - password lookup service -- responder */

#include <stdio.h>
#include <pwd.h>
#include "ryresponder.h"                /* for generic idempotent responders */
#include "PasswordLookup-ops.h"         /* operation definitions */
#include "PasswordLookup-types.h"       /* type definitions */


#define xalloc(p, type) \                                                  10
        ((p) = (type) calloc (1, sizeof *(p)))

#define salloc(s) \
        str2qb ((s), strlen (s), 1)


#ifdef  SYS5
struct passwd *getpwnam (), *getpwuid ();
#endif
                                                                           20

/* DATA */

static char *myservice = "passwdstore";

static char *mycontext = "isode passwd lookup demo";


                                        /* OPERATIONS */
int     op_lookupUser (), op_lookupUID ();                                 30

static struct dispatch dispatches[] = {
```

```
    if (q == NULL)
        return;

    for (p = q -> qb_forw; p != q; p = p -> qb_forw)
        printf ("%*.*s", p -> qb_len, p -> qb_len, p -> qb_data);
}

/* ERRORS */

/* ARGSUSED */                                                    200

static int  lookup_error (sd, id, error, parameter, roi)
int     sd,
        id,
        error;
caddr_t parameter;
struct RoSAPindication *roi;
{
    register struct RyError *rye;
                                                                  210
    if (error == RY_REJECT) {
        advise (NULLCP, "%s", RoErrString ((int) parameter));
        return OK;
    }

    if (rye = finderrbyerr (table_PasswordLookup_Errors, error))
        advise (NULLCP, "%s",  rye -> rye_name);
    else
        advise (NULLCP, "Error %d", error);
                                                                  220
    return OK;
}
```

Figure 11.3: Initiator for the Password Lookup Service

```
        adios (NULLCP, "release rejected by peer:  %d", acr -> acr_reason);
    }

    ACRFREE (acr);

    exit (0);
}

/* RESULTS */                                                    160

/* ARGSUSED */

static int  lookup_result (sd, id, dummy, result, roi)
int         sd,
            id,
            dummy;
register struct type_PasswordLookup_Passwd *result;
struct RoSAPindication *roi;
{                                                                170
    print_qb (result -> name);
    putchar (':');
    print_qb (result -> passwd);
    printf (":%d:%d:", result -> uid -> parm, result -> gid -> parm);
    print_qb (result -> gecos);
    putchar (':');
    print_qb (result -> dir);
    putchar (':');
    print_qb (result -> shell);
    putchar ('\n');                                              180

    return OK;
}


static    print_qb (q)
register struct qbuf *q;
{
    register struct qbuf *p;
                                                                 190
```

```
/*  */

/* ARGSUSED */

static int  do_help (sd, ds, args, dummy)
int         sd;
register struct dispatch *ds;                                   120
char  **args;
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;
}
                                                                130
/*  */

/* ARGSUSED */

static int  do_quit (sd, ds, args, dummy)
int         sd;
struct dispatch *ds;
char  **args;
caddr_t *dummy;
{                                                               140
    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) {                              150
        (void) AcUAbortRequest (sd, NULLPEP, 0, aci);
```

```
{
    char   *cp;

    if ((cp = *args++) == NULL) {
        advise (NULLCP, "usage:  lookupUser username");
        return NOTOK;
    }

    if ((*arg = str2qb (cp, strlen (cp), 1)) == NULL)
        adios (NULLCP, "out of memory");

    return OK;
}

/*  */

/* ARGSUSED */

static int  do_lookupUID (sd, ds, args, arg)
int         sd;
struct dispatch *ds;
char  **args;
register struct type_PasswordLookup_UserID **arg;
{
    char   *cp;

    if ((cp = *args++) == NULL) {
        advise (NULLCP, "usage:  lookupUID userid");
        return NOTOK;
    }

    if ((*arg = (struct type_PasswordLookup_UserID *) calloc (1, sizeof **arg))
            == NULL)
        adios (NULLCP, "out of memory");

    (*arg) -> parm = atoi (cp);

    return OK;
}
```

```
    "find user by id",

    "help", 0,
    do_help, NULLIFP,
    NULLIFP, NULLIFP,
    "print this information",

    "quit", 0,
    do_quit, NULLIFP,
    NULLIFP, NULLIFP,
    "terminate the association and exit",

    NULL
};

/* MAIN */

/* ARGSUSED */

main (argc, argv, envp)
int       argc;
char  **argv,
     **envp;
{
    (void) ryinitiator (argc, argv, myservice, mycontext, mypci,
                        table_PasswordLookup_Operations, dispatches, do_quit);

    exit (0);                       /* NOTREACHED */
}

/* ARGUMENTS */

/* ARGSUSED */

static int  do_lookupUser (sd, ds, args, arg)
int       sd;
struct dispatch *ds;
char  **args;
register struct type_PasswordLookup_UserName **arg;
```

11.3 Interactive Initiator

The interactive initiator of the service is placed in *lookup.c*. This file is shown in Figure 11.3 starting on page 169.

```
/* lookup.c - password lookup service -- initiator */

#include <stdio.h>
#include "ryinitiator.h"                /* for generic interctive initiators */
#include "PasswordLookup-ops.h"         /* operation definitions */
#include "PasswordLookup-types.h"       /* type definitions */

/* DATA */

static char *myservice = "passwdstore";                                    10

static char *mycontext = "isode passwd lookup demo";
static char *mypci = "isode passwd lookup demo pci";


                                        /* ARGUMENTS */
int     do_lookupUser (), do_lookupUID (), do_help (), do_quit ();

                                        /* RESULTS */
int     lookup_result ();                                                  20

                                        /* ERRORS */
int     lookup_error ();


static struct dispatch dispatches[] = {
    "lookupUser", operation_PasswordLookup_lookupUser,
    do_lookupUser, free_PasswordLookup_UserName,
    lookup_result, lookup_error,
    "find user by name",                                                   30

    "lookupUID", operation_PasswordLookup_lookupUID,
    do_lookupUID, free_PasswordLookup_UserID,
    lookup_result, lookup_error,
```

```
# clean

clean:;         rm -f *.o PasswordLookup* lookupd lookup z* _* core      120

grind:;         iprint READ-ME Makefile
                tgrind -lc $(HFILES) $(CFILES)
                tgrind -lpepy -d ../../pepy/grindefs $(RYFILES)
                @echo $(MANUALS) | \
                        tr " " "\012" | \
                        sed -e "s%.*%itroff -man &%" | \
                        sh -ve

true:;                                                                   130
```

Figure 11.2: Makefile for the Password Lookup Service

```
$(BINDIR)lookup:        lookup
                -cp $@ zlookup                                          80
                -rm -f $@
                cp lookup $@
                -@ls -gls $@
                -@echo ""

lookup:         lookup.o PasswordLookup-Iops.o \
                        ryinitiator.o PasswordLookup_tables.o
                $(LDCC) $(LDFLAGS) -o $@ lookup.o \
                        PasswordLookup-Iops.o \
                        ryinitiator.o \                                 90
                        PasswordLookup_tables.o \
                        $(LIBES)

l-lookup:       PasswordLookup-ops.c PasswordLookup_tables.c true
                $(LINT) $(LFLAGS) -DINVOKER lookup.c \
                        PasswordLookup-ops.c ryinitiator.c \
                        PasswordLookup-stubs.c \
                        PasswordLookup_tables.c $(LLIBS)

lookup.o:       ryinitiator.h PasswordLookup-ops.h PasswordLookup-types.h

PasswordLookup-Iops.o:  PasswordLookup-ops.c PasswordLookup-ops.h
                $(CC) $(CFLAGS) -DINVOKER -c PasswordLookup-ops.c
                mv PasswordLookup-ops.o $@

ryinitiator.o:  ryinitiator.h


# manual pages
                                                                        110
MANUALS =       lookupd.8c lookup.1c

manuals:;       install -c -m 0444 lookupd.8c \
                        /usr/man/man8/lookupd.8c
                install -c -m 0444 lookup.1c /usr/man/mann/lookup.n
```

```
		-@echo ""

lookupd:	lookupd.o PasswordLookup-Rops.o \
			ryresponder.o PasswordLookup_tables.o
		$(LDCC) $(LDFLAGS) -o $@ lookupd.o \
			PasswordLookup-Rops.o \
			ryresponder.o
			PasswordLookup_tables.o \
			$(LIBES)

l-lookupd:	PasswordLookup-ops.c PasswordLookup_tables.c true
		$(LINT) $(LFLAGS) -DPERFORMER lookupd.c \
			PasswordLookup-ops.c ryresponder.c \
			PasswordLookup_tables.c $(LLIBS)

lookupd.o:	ryresponder.h PasswordLookup-ops.h \
			PasswordLookup-types.h

PasswordLookup-Rops.o: PasswordLookup-ops.c PasswordLookup-ops.h
		$(CC) $(CFLAGS) -DPERFORMER -c PasswordLookup-ops.c
		mv PasswordLookup-ops.o $@

ryresponder.o:	ryresponder.h

PasswordLookup_tables.o: PasswordLookup_tables.c PasswordLookup-types.h

PasswordLookup_tables.c PasswordLookup-types.h:  PasswordLookup-asn.py
		pepsy -A -f -h -m PasswordLookup-asn.py

PasswordLookup-asn.py:  lookup.ry
		rosy -m -pepsy -o $@ lookup.ry
PasswordLookup-ops.c:   lookup.ry
PasswordLookup-ops.h:   lookup.ry
PasswordLookup-stubs.c: lookup.ry


# lookup

inst-lookup:	$(BINDIR)lookup
```

```
# Instructions to Make, for compilation of ROS-based password lookup service

BINDIR =        /usr/new/
SBINDIR =       /usr/etc/
ETCDIR =        /usr/etc/

CC      =       /bin/cc
CFLAGS =        -g   $(OPTIONS)
LINT    =       lint
LFLAGS =        -bhuz $(OPTIONS)                                    10
LLIBS   =       -lisode
LD      =       ld
LDCC    =       /bin/cc
LDFLAGS =       -g
LIBES   =       -lisode

HFILES =        ryinitiator.h ryresponder.h
CFILES =        lookupd.c lookup.c ryinitiator.c ryresponder.c
RYFILES =       lookup.ry
                                                                    20

# Here it is...

all:            lookupd lookup
inst-all:       inst-lookupd inst-lookup # manuals
install:        inst-all clean
lint:           l-lookupd l-lookup


# lookupd                                                           30

inst-lookupd:   $(SBINDIR)ros.lookup

$(SBINDIR)ros.lookup:   lookupd
                -cp $@ zros.lookup
                -rm -f $@
                cp lookupd $@
                -@ls -gls $@
```

```
GroupID ::=
   [APPLICATION 4]                                          80
        IMPLICIT INTEGER

END
```

Figure 11.1: Remote Operations Module for the Password Lookup Service

```
        passwd[1]
                IMPLICIT IA5String
                OPTIONAL,

        uid[2]
                IMPLICIT UserID,

        gid[3]
                IMPLICIT GroupID,

        quota[4]
                IMPLICIT INTEGER
                DEFAULT 0,

        comment[5]
                IMPLICIT IA5String
                OPTIONAL,

        gecos[6]
                IMPLICIT IA5String
                OPTIONAL,

        dir[7]
                IMPLICIT IA5String
                OPTIONAL,

        shell[8]
                IMPLICIT IA5String
                OPTIONAL
    }

UserName ::=
    [APPLICATION 2]
        IMPLICIT GraphicString

UserID ::=
    [APPLICATION 3]
        IMPLICIT INTEGER
```

```
PasswordLookup DEFINITIONS ::=

BEGIN

-- operations

                        -- given a user name, return a Passwd type
lookupUser OPERATION
        ARGUMENT    UserName
        RESULT      Passwd                                          10
        ERRORS      { noSuchUser, congested }
        ::=         0

                        -- given a user ID, return a Passwd type
lookupUID OPERATION
        ARGUMENT    UserID
        RESULT      Passwd
        ERRORS      { noSuchUser, congested }
        ::=         1
                                                                    20
-- errors

                        -- no matching user in the database
noSuchUser ERROR
        ::=         0

                        -- congestion at responder
congested ERROR
        ::=         1
                                                                    30
-- types

                        -- similar to an entry in <pwd.h>
Passwd ::=
   [APPLICATION 1]
        IMPLICIT SEQUENCE {
           name[0]
                IMPLICIT UserName,
```

Directory entry: if we select `p` as the lowest unassigned TSAP ID between `1024` and `2047` inclusive, e.g., `1040`, and decide to name the service as `passwdstore`, we have:

```
objectClass= top & applicationEntity & quipuObject & iSODEApplicationEntity
cn= passwdstore
presentationAddress= #1040/Internet=mydomainame
supportedApplicationContext= isode passwd lookup
acl=
execVector= ros.lookup
```

11.2 Remote Operations Module

The remote operations module for the service is placed in *lookup.ry*. This file shown in Figure 11.1 starting on page 159.

The *Makefile* for the service is shown in Figure 11.2 starting on page 163.

Chapter 11

A Complete Example

Now, let's consider a complete example. Our application is modeled after the example given in [ECoop82]. In short, we wish to implement a remote lookup facility of the UNIX */etc/passwd* database.

11.1 Service Definitions

We begin by defining the mundane naming and addressing information for the service. Using the "standard" approach described in Section 3.1, we select:

abstract syntax: defined in the *isobjects*(5) file as:

```
"local service pci"      1.17.2.n.1
```

If we select `n` as the lowest unassigned number in the `1.17.2` subtree, e.g., `7`, we might have:

```
"isode passwd lookup pci"      1.17.2.7.1
```

application context name: defined in the *isobjects*(5) file as:

```
"local service"          1.17.2.n.2
```

Similarly, for a value of `7` for `n`, we have:

```
"isode passwd lookup"          1.17.2.7.2
```

10.1.4 An Example

An example of a responder written using this boilerplate is shown in Section 11.4 on page 170.

```
void    adios (what, fmt)
char   *what,
       *fmt;
{
    adios (what, fmt);
}
#endif                                                          70


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

    va_start (ap);                                              80

    code = va_arg (ap, int);

    _ll_log (pgm_log, code, ap);

    va_end (ap);
}
#else
/* VARARGS3 */
                                                                90
void    advise (code, what, fmt)
char   *what,
       *fmt;
int     code;
{
    advise (code, what, fmt);
}
#endif
```

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


void    acs_advise (aca, event)
register struct AcSAPabort *aca;                                        30
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));   40

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


#ifndef lint
void    adios (va_alist)
va_dcl
{                                                                       50
    va_list ap;

    va_start (ap);

    _ll_log (pgm_log, LLOG_FATAL, ap);

    va_end (ap);

    _exit (1);
}                                                                       60
#else
/* VARARGS2 */
```

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);
}                                                                    10


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),20
                rop -> rop_cc, rop -> rop_cc, rop -> rop_data);
    else
        (void) sprintf (buffer, "[%s]", RoErrString (rop -> rop_reason));
```

```
        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
}

...
```

```
/* 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)
```

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 100. 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);
```

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

...
```

```
            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,
```

```
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");
            }
```

```
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;                170

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

        default:
            if (stopfnx)
                (*stopfnx) (fd, (struct AcSAPfinish *) 0);
        case DONE:
            (void) AcUAbortRequest (fd, NULLPEP, 0, &acis);                   180
            (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:                                                            190
            ros_indication (fd, roi);
            break;

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

    return OK;
}
                                                                              200
```

```
        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
```

```
            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");
```

```
        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)
```

```
#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,
```

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"
```

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 27 of *Volume One*). The routine `AcAssocResponse` is then called to deal with the incoming association. The arguments are strictly boilerplate: they will work

```
/* 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;
                                                                    10
    IFP     ds_vector;
};


extern int  debug;


void    adios (), advise ();
void    acs_advise ();
void    ros_adios (), ros_advise ();                                20

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;
```

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.

```
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);
}
```

9.2.3 An Example

An example of an interactive initiator written using this boilerplate is shown in Section 11.3 on page 164. 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)
```

```
    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
```

```
                                                                    300
    va_start (ap);

    _advise (ap);

    va_end (ap);

    _exit (1);
}
#else
/* VARARGS */                                                       310

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


                                                                    320
#ifndef lint
void      advise (va_alist)
va_dcl
{
    va_list ap;

    va_start (ap);

    _advise (ap);
                                                                    330
    va_end (ap);
}


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

```
/*  */

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

    _exit (1);
}


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

/*  */

#ifndef lint
void    _advise ();


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

```
            return NOTOK;
        }

        if (cp < ep)
            *cp++ = i;
    }
    *cp = NULL;

    return OK;
}

/*  */

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);
}
```

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

        default:
            adios (NULLCP, "unknown return from RyStub=%d", result);
            /* NOTREACHED */
    }                                                                   190

    if (ds -> ds_free && in)
        (void) (*ds -> ds_free) (in);
}

/*  */

static int  getline (buffer)
char   *buffer;
{                                                                       200
    register int    i;
    register char  *cp,
                   *ep;
    static int  sticky = 0;

    if (sticky) {
        sticky = 0;
        return NOTOK;
    }
                                                                        210
    printf ("%s> ", myname);
    (void) fflush (stdout);

    for (ep = (cp = buffer) + BUFSIZ - 1; (i = getchar ()) != '\n';) {
        if (i == EOF) {
            printf ("\n");
            clearerr (stdin);
            if (cp != buffer) {
                sticky++;
                break;                                                  220
            }
```

```
            invoke (sd, ops, ds, vec + 1);
        }
    }
    else
        invoke (sd, ops, ds, argv + 3);
                                                                    150
    (*quit) (sd, (struct dispatch *) NULL, (char **) NULL, (caddr_t *) NULL);
}

/*  */

static  invoke (sd, ops, ds, args)
int     sd;
struct RyOperation ops[];
register struct dispatch *ds;
char  **args;                                                       160
{
    int     result;
    caddr_t in;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    in = NULL;
    if (ds -> ds_argument && (*ds -> ds_argument) (sd, ds, args, &in) == NOTOK)
        return;                                                     170

    switch (result = RyStub (sd, ops, ds -> ds_operation, RyGenID (sd), NULLIP,
                    in, ds -> ds_result, ds -> ds_error, ROS_SYNC, roi)) {
        case NOTOK:             /* failure */
            if (ROS_FATAL (rop -> rop_reason))
                ros_adios (rop, "STUB");
            ros_advise (rop, "STUB");
            break;

        case OK:                /* got a result/error response */   180
            break;
```

```
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");
                                                                        110
if (acc -> acc_result != ACS_ACCEPT) {
        if (iloop)
            printf ("failed\n");

        adios (NULLCP, "association rejected:  [%s]",
                AcErrString (acc -> acc_result));
}
if (iloop) {
        printf ("connected\n");
        (void) fflush (stdout);                                         120
}

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

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

if (iloop) {
        for (;;) {                                                      130
            if (getline (buffer) == NOTOK)
                break;

            if (str2vec (buffer, vec) < 1)
                continue;

            for (ds = dispatches; ds -> ds_name; ds++)
                if (strcmp (ds -> ds_name, vec[0]) == 0)
                    break;
            if (ds -> ds_name == NULL) {                                140
                advise (NULLCP, "unknown operation \"%s\"", vec[0]);
                continue;
            }
```

```
if ((pci = ode2oid (mypci)) == NULLOID)
        adios (NULLCP, "%s:  unknown object descriptor", mypci);
if ((pci = oid_cpy (pci)) == NULLOID)
        adios (NULLCP, "out of memory");
pc -> pc_nctx = 1;                                                      70
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);
}

if (argc < 3) {                                                         80
        printf ("%s", myname);
        if (sf -> sr_ulen > 2)
            printf (" running on host %s", sf -> sr_udata + 2);
        if (sf -> sr_clen > 2)
            printf (" at %s", sf -> sr_cdata + 2);
        printf (" [%s, ", oid2ode (ctx));
        printf ("%s]\n", oid2ode (pci));
        printf ("using %s\n", isodeversion);

        printf ("%s...  ", argv[1]);                                    90
        (void) fflush (stdout);

        iloop = 1;
}
else {
        for (ds = dispatches; ds -> ds_name; ds++)
            if (strcmp (ds -> ds_name, argv[2]) == 0)
                break;
        if (ds -> ds_name == NULL)
            adios (NULLCP, "unknown operation \"%s\"", argv[2]);        100

        iloop = 0;
}
```

```
        sd;
char    buffer[BUFSIZ],
        *vec[NVEC + 1];
register struct dispatch   *ds;                                    30
struct SSAPref sfs;
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,                                                       40
        pci;
struct PSAPctxlist pcs;
register struct PSAPctxlist *pc = &pcs;
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)                             50
    myname = argv[0];

if (argc < 2)
    adios (NULLCP, "usage:  %s host [operation [ arguments ...  ]]", myname);

if ((aei = _str2aei (argv[1], myservice, mycontext, argc < 3,
                     NULLCP, NULLCP)) == NULLAEI)
    adios (NULLCP, "unable to resolve service:  %s", PY_pepy);
if ((pa = aei2addr (aei)) == NULLPA)
    adios (NULLCP, "address translation failed");                  60

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

The routine `AcAssocRequest` is called to establish an association for the service. If successful, the routine `RoSetService` is used to tell the remote operations library to use the presentation service as the underlying service.

The interactive loop is then entered: a line is read from the standard input (the `getline` routine is called), it is broken into components, a search is performed on the dispatch table to find a matching keyword, and then finally the user-supplied routine is called. When end-of-file is reached (or if the user-supplied routine returns the manifest constant `DONE`), the assocation is released.

If additional arguments were present on the invocation line, then the named operation is invoked instead of entering an interactive loop.

```
/* ryinitiator.c - generic interactive initiator */

#include <stdio.h>
#include <varargs.h>
#include "ryinitiator.h"

/*    DATA */

static char *myname = "ryinitiator";

extern char *isodeversion;

/*    INITIATOR */

ryinitiator (argc, argv, myservice, mycontext, mypci, ops, dispatches, quit)
int         argc;
char  **argv,
      *myservice,
      *mycontext,
      *mypci;
struct RyOperation ops[];
struct dispatch *dispatches;
IFP         quit;
{
   int         iloop,
```

9.2.2 Worker Routines

Now, let's consider the routines which implement the interactive initiator. There are only two: `ryinitiator` and `getline`.

The `ryinitiator` routine does most of the work:

```
int     ryinitiator (argc, argv, myservice, mycontext,
                mypci, ops, dispatches, quit)
int     argc;
char  **argv,
       *myservice,
       *mycontext,
       *mypci;
struct RyOperation ops[];
struct dispatch *dispatches;
IFP    quit;
```

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;

`mycontext`: the application context name of the service;

`mypci`: the abstract syntax of the service;

`ops`: the operations defined for the service;

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

`quit`: a routine to call when the association is to be terminated.

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 rudimentary argument check is done, and the application-entity information and presentation address for the desired service are computed. This is followed by computing the application context and default presentation context information for the service, along with a session reference identifier. Finally, some diagnostic information may be printed out if the program will operate in interactive mode.

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 ();
```

```
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
```

```
                                                                    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);
}
```

```
                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;
char    *event;
{
    acs_advise (aca, event);

    _exit (1);
}


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


#ifndef lint
void    _advise ();
```

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),
```

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);

...
```

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

    return OK;
}

...
```

```
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:
```

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);

...
```

```
            /* NOTREACHED */

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

    free_MODULE_argument (arg);

...
```

```
                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");
```

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 94. 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))
```

```
    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");

...
```

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);
```

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

`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`.

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:

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.

8.7 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.

```
        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 58 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.

The return value of the dispatch routine is consulted to determine the disposition of the invocation. One of three values should be returned: `NOTOK`, which indicates that some non-operational error occured; `OK`, which indicates that the operation was handled; or, `DONE`, which indicates that the association is pending termination.

8.6.2 Responding to Operations

If the operation corresponding to the dispatch routine does not return a result, then the routine may simply process the request and return. Otherwise, the dispatch routine should ultimately cause one of three routines to be called to handle an invocation.

The routine `RyDsResult` is used to return a result to an invocation.

```
int     RyDsResult (sd, id, out, priority, roi)
int     sd;
int     id,
        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;

`out`: the address of the *C* structure for the operation's result (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 `RyDsError` is used to return an error to an invocation.

```
int     RyDsError (sd, id, err, out, priority, roi)
int     sd;
int     id,
```

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

`op`: the operation code of the operation to be registered;

`fnx`: the address of a dispatch routine to be invoked when the operation is invoked; and,

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

An operation may be un-registered by using `NULLIFP` for the `fnx` parameter.

When a registered operation is invoked (from the routine `RyWait` described in Section 8.7), the dispatch routine is invoked with five parameters:

```
result = (*fnx) (sd, ryo, rox, in, roi)
int     sd;
struct RyOperation *ryo;
struct RoSAPinvoke *rox;
caddr_t in;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`ryo`: a pointer to a entry in the `RyOperation` table for this operation;

`rox`: a pointer to a `RoSAPinvoke` structure containing miscellaneous information regarding the operation being invoked (e.g., the invocation id);

`in`: the address of the *C* structure for the operation's argument (if any); and,

`roi`: a pointer to a `RoSAPindication` structure, which is updated only if the operation was not performed.

Note that if an argument is present for the operation, then the *librosy*(3n) library will automatically release the memory used by the *C* structure when the dispatch routine returns.

```
        id;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association descriptor;

`sd`: the invocation-identifier associated with the operation; and,

`roi`: a pointer to a `RoSAPindication` structure, which is updated only on failure.

The `RyDiscard` routine returns either `NOTOK` (on failure) or `OK` (on success).

8.5 Routines for Responders

Responders use the routines in the *libacsap*(3n) library for association control. These are described in Chapter 2 of *Volume One*.

8.6 Routines for Performers

After forming an association, performers register the operations they are willing to perform. They then typically wait for those operations to be invoked.

8.6.1 Registering Operations

The routine `RyDispatch` is used to register a handler for an operation:

```
int     RyDispatch (sd, ryo, op, fnx, roi)
int     sd;
struct RyOperation *ryo;
int     op;
IFP     fnx;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`sd`: the association-descriptor;

`id`: the invocation identifier associated with the result or error;

`reason`: either `RY_RESULT` if the result dispatch routine is invoked, the error code if the error dispatch routine is invoked with an error, or `RY_REJECT` if the error dispatch routine is invoked with a rejection;

`value`: the address of the *C* structure for the result or error parameter (if any); and,

`roi`: a pointer to a `RoSAPindication` structure, which is updated only if the result, error, or rejection is faulty.

The dispatch routine should return the manifest constant `OK` unless you know what you're doing. Note that if `value` is present, then the *librosy*(3n) library will automatically release the memory used by the *C* structure when the dispatch routine returns.

If the call to `RyStub` 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, as this normally does not happen to initiators.

If the value of the `class` parameter was `ROS_INTR`, then `RyStub` issues a synchronous call (as described above for the `ROS_SYNC` case). However, if the user generates an interrupt, usually by typing control-C (`^C`), while waiting for the response to the operation, then `RyStub` will return immediately with the value `NOTOK` and the `rop_reason` element of the `RoSAPpreject` structure contained in the `RoSAPindication` parameter `roi` will be set to the value `ROS_INTERRUPTED`.

At this time, the program may do other processing and later return to wait for the outcome of the operation. Instead, the program may direct the run-time environment to silently discard any outcome. This is useful when the user wishes to abandon an inprogress operation and really doesn't care what the results are (i.e., when reading from a database). The verb"RyDiscard" routine is used to accomplish this function:

```
int     RyDiscard (sd, id, roi)
int     sd,
```

`linked`: the linked invocation identifier to use, if any (use `NULLIP` if this is not a linked reply);

`in`: the address of the *C* structure for the operation's argument;

`rfx`: the address of a dispatch routine to be invoked if the operation returns a result;

`efx`: the address of a dispatch routine to be invoked if the operation returns an error or is rejected;

`class`: the operation class (either `ROS_SYNC` for a synchronous operation, `ROS_ASYNC` for an asynchronous operation, or `ROS_INTR` for a synchronous but user-interruptable operation); and,

`roi`: a pointer to a `RoSAPindication` structure, which is updated only if the operation was not performed.

The `RyStub` routine returns one of three values: `NOTOK` (on failure), `OK` (on success), or `DONE` (when something else happens).

If the call to `RyStub` 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 `RyStub` returns the manifest constant `OK`, then the operation was queued or completed, depending on the value of the `class` parameter. If the value of the `class` parameter was `ROS_ASYNC`, then `RyStub` returns immediately after queuing the invocation. Otherwise, `RyStub` waits for a response to the operation, invokes the appropriate dispatch routine, and then returns. Both the result and error dispatch routines are invoked in the same way:

```
result = (*fnx) (sd, id, reason, value, roi)
int     sd;
int     id,
        reason;
caddr_t value;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`class`: `ROS_SYNC`;

`invokeID`: a unique-number;

`linkedID`: `NULLIP`; and,

`priority`: `ROS_NOPRIO`.

Asynchronous Interface

Although most invokers desire a simple interface for synchronous execution, it is also useful to have an asynchronous interface. Further, since a synchronous interface can be thought of as a degenerate subset of an asynchronous interface, most invokers use the asynchronous interface and simply ask for it to block after making the invocation. The `RyStub` routine is used to provide such an interface.

```
int     RyStub (sd, ryo, op, id, linked, in, rfx, efx,
                class, roi)
int     sd;
struct RyOperation *ryo;
int     op,
        id,
       *linked,
        class;
caddr_t in;
IFP     rfx,
        efx;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

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

`op`: the operation code of the operation to invoke;

`id`: the invocation identifier to use;

```
IFP     rfx,
        efx;
caddr_t in,
       *out;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

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

`op`: the operation code of the operation to invoke;

`in`: the address of the *C* structure for the operation's argument;

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

`rfx`: the address of a dispatch routine to be invoked if the operation returns a result;

`efx`: the address of a dispatch routine to be invoked if the operation returns an error;

`class`: the operation class (either `ROS_SYNC` for a synchronous operation, or `ROS_ASYNC` for an asynchronous one);

`invokeID`: the invocation ID of this request;

`linkedID`: the linked ID of this request (use `NULLIP` if not linked);

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

`roi`: a pointer to a `RoSAPindication` structure, which is updated only if the operation was not performed.

This routine is exactly equivalent to the `RoInvokeRequest` routine in the *librosap*(3n) library, with the additional functions of: automatically encoding and decoding arguments, results, and parameters. Hence, the routine `RyOperation`, is essentially a call to `RyOpInvoke` with a few hard-wired parameters:

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

`roi`: a pointer to a `RoSAPindication` structure, which is updated only if the operation was not performed.

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

If the call to `RyOperation` 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 `RyOperation` returns the manifest constant `OK`, then the `response` and `out` parameters are updated. If the value of the integer pointed to by `response` is `RY_RESULT`, then the value of the address pointed to by `out` is a newly allocated *C* structure for the operation's result (if any). If the value of the integer pointed to by `response` is any other value, then it indicates the error which the operation returned, and the value of the address pointed to by `out` is a newly allocated *C* structure for the 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 `RyOperation` 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, as this normally does not happen to initiators.

The routine `RyOperation` is simply a comprehensive interface for a more general routine, `RyOpInvoke`, which invokes an operation:

```
int     RyOpInvoke (sd, ryo, op, in, out, rfx, efx, class,
                    invokeID, linkedID, priority, roi)
int     sd;
struct RyOperation *ryo;
int     op,
        class,
        invokeID,
       *linkedID,
        priority;
```

8.3 Routines for Initiators

Initiators use the routines in the *libacsap*(3n) library for association control. These are described in Chapter 2 of *Volume One*.

8.4 Routines for Invokers

After forming an association, invokers are able to request operations for execution by the performer. Two interfaces are provided: a synchronous interface and an asynchronous interface.

Synchronous Interface

Most invokers desire a simple interface for synchronous execution of operations (the operation blocks until it is completed). The `RyOperation` routine is used to provide such an interface.

```
int     RyOperation (sd, ryo, op, in, out, response, roi)
int     sd;
struct RyOperation *ryo;
int     op,
       *response;
caddr_t in,
       *out;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

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

`op`: the operation code of the operation to invoke;

`response`: an integer-valued location which is updated if the call succeeds;

`in`: the address of the *C* structure for the operation's argument;

`ryo_arg_free`: the routine to free the *C* structure created by the argument decode routine for the operation;

`ryo_result`: non-zero if this operation returns a result;

`ryo_res_encode/ryo_res_decode`: the address of the routines to encode or decode (respectively) the operation's result from a *C* structure to its corresponding abstract syntax (present only if the operation returns a result);

`ryo_res_free`: the routine to free the *C* structure created by the result decode routine for the operation; and,

`ryo_errors`: a pointer to null-terminated list of the errors which this operation may generate (present only if the operation may returns at least one error).

The `RyError` structure is used to define the compiled parts of an error:

```
struct RyError {
    char   *rye_name;
    int     rye_err;

    IFP     rye_param_encode;
    IFP     rye_param_decode;
    IFP     rye_param_free;
};
```

The elements of this structure are:

`rye_name/rye_err`: the error name and code;

`rye_param_encode/rye_res_decode`: the address of the routines to encode or decode (respectively) the error's parameter from a *C* structure to its corresponding abstract syntax (present only if the error contains a parameter); and,

`ryo_param_free`: the routine to free the *C* structure created by the parameter decode routine for the error.

The routines all work on an established association, which is referenced by an association-descriptor. This is usually the first parameter given to any of the routines in the *librosy*(3n) library. Further, the last parameter is usually a pointer to a `RoSAPindication` structure. If a call to one of these routines fails, then the structure is updated. Section 3.3 of *Volume One* describes this structure and how to interpret error conditions and their severity.

8.2.1 Interface from ROSY

For a given module, e.g., `MODULE`, *rosy* will define two tables:

```
struct RyOperation table_MODULE_Operations[];

struct RyError table_MODULE_Errors[];
```

The `RyOperation` structure is used to define the compiled parts of an operation:

```
struct RyOperation {
    char   *ryo_name;
    int     ryo_op;

    IFP     ryo_arg_encode;
    IFP     ryo_arg_decode;
    IFP     ryo_arg_free;

    int     ryo_result;
    IFP     ryo_res_encode;
    IFP     ryo_res_decode;
    IFP     ryo_res_free;

    struct RyError **ryo_errors;
};
```

The elements of this structure are:

`ryo_name/ryo_op`: the operation name and code;

`ryo_arg_encode/ryo_arg_decode`: the address of the routines to encode or decode (respectively) the operation's argument from a *C* structure to its corresponding abstract syntax (present only if the operation takes an argument);

Chapter 8

Run-Time Environment

The *librosap*(3n) library implements the run-time environment for the stubs generated by the *rosy*(1) program.

The routines described herein provide a more structured interface to the remote operations service (ROS). They are used in the boilerplate routines which are described in the following chapters. Hence, the more hasty reader may wish to skip this chapter on first reading of this volume.

8.1 Notice

Please read the following important message.

> **NOTE:** When using the *librosy*(3n) library, no calls should be made to the *librosap*(3n) library. (That is, the `RoXXX` routines should not be directly referenced.)
> There are three exceptions: `RoSetService` and its related arguments (e.g., `RoPService`), `RoSelectMask`, and `RoErrString`.

8.2 Conventions

All of the routines in the *librosy*(3n) library are integer-valued. Unless otherwise indicated, they return the manifest constant `OK` on success, or `NOTOK` otherwise.

Part IV

Recipes

The '-d' switch is a no-op with *pepsy*, as it always ignores all *pepy*-style augmentations, including the "verbatim" actions occuring at the very beginning and end of the module.

The '-f' switch directs *posy* to generate *C* macros to deallocate the structures it defines (e.g., for type `type_MODULE_type`, a macro called `free_MODULE_type` is defined).

As with *posy*, the '-h' switch enables additional heuristics when *pepsy* generates a *C* language structure definition. See Section 5.4.1 on page 50.

Normally, *posy* prints the name of each type as it works. The '-s' switch disables this behavior.

7.4.2 Makefiles

By convention, input files to *pepsy* have the extension `.py`. In order to remain consistent with the naming scheme used by *posy*, rather than starting with file `module.py`, we assume we are starting with file `MODULE-asn.py`. Let us arbitrarily select the file named `MODULE-types.py` for the augmented abstract syntax module produced by *posy*. Hence

```
pepsy MODULE-asn.py
```

will produce:

`MODULE-types.h`: the *C* language structures; and,

`MODULE_tables.c`: the mapping tables.

Our *make*(1) rules are:

```
MODULE_tables.o:  MODULE_tables.c MODULE-types.h

MODULE_tables.c MODULE_tables.h: MODULE-asn.py
pepsy -A -f -h MODULE-asn.py
```

7.2 PEPSY Environment

The *pepsy* program will produce two files after reading its abstract syntax module.

7.2.1 C Language Structures

For each type in the abstract syntax module, *pepy* will define a corresponding *C* language structure (`struct`), using the same rules as *posy* (see Section 5.2.1 on page 41).

7.2.2 Mapping tables

In addition to the definition of *C* language structures, *pepsy* will generate a *C* file containing several tables that define the mapping between the structures and the corresponding ASN.1 objects. As this is entirely an automatic process, the contents of the tables are relatively unimportant.

7.3 Known Deficiences

pepsy uses essentially the same front-end as the *posy*(1) program, so it has some limitations in the ASN.1 syntax it can accept. Consult Section 5.3 on page 50 for the details.

7.4 Running PEPSY

Here are a few things to know when running *pepsy*.

7.4.1 Options

The *pepsy* program has a few options to modify its behavior.

The '`-A`' switch directs *pepsy* to produce mapping tables for encoding, decoding, and printing.

The '`-a`' switch directs *pepy* to augment the `#include` file it generates with commentary text.

Chapter 7

Pepsy

The *pepsy*(1) program is a table-driven replacement for *pepy* and *posy*. At present, all three compilers are available as a part of the standard distribution. In the next major release, *pepy* and *posy* will remain, but most likely will not be installed by default.

The remainder of this chapter describes how to use *pepsy*. By virtue of having presented the preceeding two chapters, only a terse discussion is needed!

7.1 Abstract Syntax Module

As with *posy*, the *pepsy* program reads a description of an *abstract syntax module* and produces the corresponding *C* structures along with several tables that define the mapping between the ASN.1 objects in the module and the *C* structures. This tables replaces the work done by the *pepy* program.

Note that unlike *posy*, *pepsy* does not accept annotations in the abstract syntax module. It should be raw ASN.1, either entered by hand, or produced by the *rosy* program. Note that you must tell *rosy* when *pepsy* will be used. This is accomplished with the '`-pepsy`' option, e.g.,

```
.ry.py:;        rosy -pepsy $(RYFLAGS) $<
```

6.8 For Further Reading

The Abstract Syntax Notation One is specified in [ISO87]. The corresponding CCITT recommendation is defined in [CCITT88c].

```
...

/*    ERRORS */

#include <varargs.h>


/* VARARGS2 */

void        adios (what, fmt, va_alist)
char    *what,
        *fmt;
va_dcl
{
    advise (what, fmt, va_alist);

    _exit (1);
}


/* VARARGS2 */

void        advise (what, fmt, va_alist)
char    *what,
        *fmt;
va_dcl
{
    char    buffer[BUFSIZ];

    asprintf (buffer, what, fmt, va_alist);

    (void) fflush (stdout);

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

    (void) fflush (stderr);
}

%}
```

Figure 6.11: Pretty-Printer finalization (continued)

```
...

%{
/*    DEBUG */

#ifdef  DEBUG
testdebug (pe, s)
register PE pe;
register char *s;
{                                                                    10
    static int  debug = OK;
    char   *cp;
    register PS     ps;

    switch (debug) {
        case NOTOK:
            break;

        case OK:
            if ((debug = (cp = getenv ("PEPYDEBUG")) && *cp ? atoi (cp)      20
                        : NOTOK) == NOTOK)
                break;
            fprintf (stderr, "%s made with %s\n", myname, pepyid);

        default:
            fprintf (stderr, "%s\n", s);

            if ((ps = ps_alloc (std_open)) == NULLPS)
                return;
            if (std_setup (ps, stderr) != NOTOK)                         30
                (void) pe2pl (ps, pe);
            fprintf (stderr, "--------\n");
            ps_free (ps);
            break;
    }
}
#endif

...
```

Figure 6.11: Pretty-Printer finalization

```
...

Date ::=
        [APPLICATION 3]
          IMPLICIT IA5String -- YYYYMMDD --
          [[s Date]]
          %{
                int year, month, day;
                static char buffer[BUFSIZ];
                                                                    10
                if (sscanf (Date, "%4d%2d%2d", &year, &month, &day) != 3
                        || year < 1900 || month < 1 || month > 12
                        || day < 1 || day > 31)
                   adios (NULLCP, "bad date format:  %s", Date);
                (void) sprintf (buffer, "%02d %s %4d", day,
                              months[month - 1], year);
                Date = buffer;
          %}

END                                                                 20

...
```

Figure 6.10: Pretty-Printer type definitions (continued)

```
...

ChildInformation ::=
        SET {
                Name
                %{
                   printf ("\nChild Information\nName:  %s %s.  %s\n",
                         givenName, initial, familyName);
                %},
                                                                        10
           dateofBirth[0]
                Date
                %{ printf ("Date of Birth:  %s\n", Date); %}
        }

Name ::=
        [APPLICATION 1]
           IMPLICIT SEQUENCE {
                givenName
                   IA5String [[s givenName]],                           20

                initial
                   IA5String [[s initial]],

                familyName
                   IA5String [[s familyName]]
           }

EmployeeNumber ::=
        [APPLICATION 2]                                                 30
           IMPLICIT INTEGER
           %{ printf ("Employee Number:  %d\n", $$); %}

...
```

Figure 6.10: Pretty-Printer type definitions

```
...

BEGIN

PersonnelRecord
        %{ char *Title; %}
        ::=
        [APPLICATION 0]
           IMPLICIT SET {
                   Name                                                    10
                   %{
                         printf ("Name:  %s %s.  %s\n",
                                 givenName, initial, familyName);
                   %},

                title[0]
                   IA5String
                   [[s Title]]
                   %{ printf ("Title:  %s\n", Title); %},
                                                                           20
                   EmployeeNumber,

                dateOfHire[1]
                   Date
                   %{ printf ("Date of Hire:  %s\n", Date); %},

                nameOfSpouse[2]
                   Name
                   %{
                         printf ("Name of Spouse:  %s %s.  %s\n",          30
                                 givenName, initial, familyName);
                   %},

                [3]
                   %{
                         PE      cc = prim2seq ($$);
                         if (cc)
                            printf ("Number of Children:  %d\n",
                                    cc -> pe_cardinal);
                   %}                                                      40
                   IMPLICIT SEQUENCE OF ChildInformation
                   DEFAULT {}
           }
           %{ printf ("--------\n"); %}
...
```

Figure 6.9: Pretty-Printer top-level

```
...

/*    MAIN */

/* ARGSUSED */

main (argc, argv, envp)
int         argc;
char  **argv,
     **envp;                                                    10
{
    PE        pe;
    PS        ps;

    myname = *argv;
    if ((ps = ps_alloc (std_open)) == NULLPS)
            adios (NULLCP, "ps_alloc loses");
    if (std_setup (ps, stdin) == NOTOK)
            ps_adios (ps, "std_setup");
                                                               20
    for (;;) {
            if ((pe = ps2pe (ps)) == NULLPE)
                if (ps -> ps_errno)
                        ps_adios (ps, "ps2pe");
                else
                        break;

            (void) parse_PERSONNEL_PersonnelRecord (pe, 1, NULLIP, NULLVP,
                        NULLCP);
                                                               30
            pe_free (pe);
    }

    ps_free (ps);

    exit (0);
}

%}
                                                               40
...
```

Figure 6.8: Pretty-Printer initialization (continued)

```
-- personnel record pretty-printer

PERSONNEL DEFINITIONS ::=

%{
#define DEBUG

#include <stdio.h>

                                                                    10
#define ps_adios(ps, f) \
        adios (NULLCP, "%s:  %s", (f), ps_error ((ps) -> ps_errno))


static char *myname = "prpp";


void    adios ();

                                                                    20
static char *givenName;
static char *initial;
static char *familyName;
static char *Title;
static char *Date;


static char *months[] = {
   "January", "February", "March", "April", "May", "June",
   "July", "August", "September", "October", "November", "December"     30
};

...
```

Figure 6.8: Pretty-Printer initialization

```
pepy|asn1|asn.1|ASN1|ASN.1|x409|x.409|X409|X.409:\
:pb=^\d?\p\d?\:\:\=:tl:\
:bb=\dBEGIN\d|{|\[|\(:be=\dEND\d|}|\]|\):\
:cb=--:ce=--|$:sb="|':se="|':\
:kw=ANY APPLICATION BEGIN BIT BITSTRING BOOLEAN\
    CHOICE COMPONENTS DEFAULT DEFINITIONS END\
    EXTERNAL FALSE IDENTIFIER IMPLICIT INTEGER\
    NULL OBJECT OCTET OCTETSTRING OF OPTIONAL\
    PRIVATE SEQUENCE SET STRING TRUE UNIVERSAL\
MACRO NOTATION TYPE VALUE\
ABSENT BY COMPONENT DEFINED ENCRYPTED\
    ENUMERATED EXPLICIT EXPORTS FROM IMPORTS\
    INCLUDES MIN MAX PRESENT REAL SIZE TAGS\
    WITH\
ARGUMENT BIND ERROR ERRORS LINKED OPERATION\
    PARAMETER RESULT UNBIND\
ABSTRACT AS CONSUMER INVOKES OPERATIONS PAIRED\
    PORT PORTS RECURRING REFINE SUPPLIER\
    VISIBLE [C] [S]\
ATTRIBUTE CLASS CONTAIN EMPTY EQUALITY FOR\
    MATCHES MAY MULTI MULTIPLE MUST NUMBER\
    SINGLE SUBCLASS SUBSTRINGS SYNTAX USES\
CONTEXT ELEMENT ELEMENTS INITIATOR REMOTE SERVICE\
    SYNTAXES\
OBJECT-TYPE SYNTAX ACCESS STATUS\
DECODER ENCODER PRINTER SECTIONS %{ %} [[ ]] << >>:
```

Figure 6.7: Grind Entry for pretty-printing PEPY programs

6.6.4 Grinding

Figure 6.7 contains an entry for the *vgrindefs*(5) file, which may be used by the *vgrind*(1) or *tgrind*(1) facility. While this entry is not perfect, it has proven to be adequate for most purposes; e.g., the examples in this chapter were pretty-printed using this entry.

In addition, if you want to get an ASN.1 specification by removing all of the *C* augmentations from the *pepy* input file, use the '`-p`' switch. For example,

```
pepy -p < myfile.py
```

can be used to view the ASN.1 specification in the file `myfile.py`.

6.7 An Example

Let's consider how one might construct a pretty-printer for the personnel records described in Figure 6.1 on page 59. (Of course, earlier in Section 6.5 the preferred method of generating pretty-printers was described.)

First, we start the module off by defining the usual prefactory *C* code as verbatim actions. This is shown in Figure 6.8. Next, we define the `PersonnelRecord` types. The rest of the types are then defined. This is shown in Figures 6.9 and 6.10 respectively. Finally, the usual closing *C* code is included as verbatim actions, as shown in Figure 6.11.

The '`-a`' switch sets the name of the routine used by the `advise` facility (see page 72).

The '`-d`' switch directs *pepy* to ignore any action statements in the input.

The '`-h`' switch enables additional heuristics when *pepy* generates a printer.

The '`-o`' switch sets the name of the output file, which is normally derived from the name of the input file. The distinguished file "`-`" can be used to force the use of the standard output.

If the '`-o`' switch is not used, then `-b␣prefix` can be used to direct *pepy* to produce multiple output files, each starting with `prefix`.

The '`-P`' switch directs *pepy* not to include *cpp*-type line number information in the output. This is useful for debugging *pepy* (gasp!).

The '`-p`' switch directs *pepy* to remove the action states from the input file and print the resulting information on the standard output. This typically results in a file suitable for pretty-printing.

The '`-r`' switch enables the generation of "robust" code. If this option is given, then the *C* code generated will not check for unknown or extraneous objects. This is used for extensibility purposes.

Normally, *pepy* prints the name of each type as it works. The '`-s`' switch disables this behavior.

Finally, the '`-S`' switch sets the initial section-processing mode for *pepy*. The default is `-S␣DECODE`, which causes *pepy* to generate a decoder. Other values are `-S␣ENCODE` to generate an encoder; or, `-S␣PRINT` to generate a printer. The '`-A`' switch is used to set the initial mode to generate encoder, decoder, and printer routines.

6.6.3 Makefiles

By convention, input files to *pepy* have the extension `.py`. The rules for *make*(1) are:

```
.SUFFIXES:      .py .c .o

.py.o:;         pepy $(PYFLAGS) $<
                $(CC) $(CFLAGS) -c $*.c
                -rm -f $*.c

.py.c:;         pepy $(PYFLAGS) $<
```

```
/* prpp.c - Personnel Record prety-printer */

#include <stdio.h>

int     print_PERSONNEL_PersonnelRecord ();


main (argc, argv, envp)
int     argc;
char  **argv,                                                    10
     **envp;
{
   exit (PY_pp (argc, argv, envp, print_PERSONNEL_PersonnelRecord));
}
```

Figure 6.6: Simpler Pretty-Printer initialization

6.6 Compiling and Loading

Programs generated by *pepy* automatically include `<isode/psap.h>`. Typically, these programs should also be loaded with `-lisode`, although only the libraries `-lpepy`, `-lpsap`, and `-licompat` are referenced.

6.6.1 External Modules

When an external module, `MODULE`, is referenced, *pepy* reads a file which is named `MODULE.ph` for various information. The *pepy* program will consult the environment variable `$PEPYPATH` as a search-list of directories to search to find this file. The components in the path are separated by a colon (':'). If this environment variable is not set, then *pepy* consults the user's current directory followed by a system-wide area defined at compile-time.

Similarly, in addition to producing a `.c` file, *pepy* will also generate a `.ph` file containing information describing the module.

6.6.2 Options

The *pepy* program has many options to modify its behavior:

```
char   PY_pepy[BUFSIZ];
```

for easy access. Note that the routines in the *libpepy*(3) library are generated to use the `PY_advise` routine. This routine is also encouraged for use when building *rosy* encoder/decoder routines.

6.4.3 Debug Output

If a program generated by *pepy* is compiled with the `-DEBUG` option, then prior to fiddling with each instance of a type, the routine `testdebug` is called.

```
int     testdebug (pe, s)
PE      pe;
char   *s;
```

The parameters to this procedure are:

- the presentation element being examined; and,
- a short null-terminated diagnostic string describing the instance of the type.

By convention, if the program was compiled with `-DDEBUG`, then the `testdebug` routine (supplied by the user) should examine the `$PEPYDEBUG` environment variable. If this variable is set to a non-zero decimal number, then it should print out debugging information. Consult Figure 6.11 on page 83 for an example of the `testdebug` procedure.

In addition, the static character pointer `pepyid` contains information on the version of *pepy* which generated the program.

6.5 Pretty-printers

To facilitate the automatic generation of pretty-printers, the routine `PY_pp` is included in the *libpepy*(3) library. Given an invocation from a simple main routine, `PY_pp` will parse invocation arguments, build presentation elements (from either a presentation stream or presentation list), and then invoke the pretty-printer routine for each presentation element. By simply invoking pepy with the '`-S␣PRINT`' option. the pretty-printer routine can be generated automatically. Figure 6.6 shows the required initialization code for the pretty-printer.

To invoke an encoding routine, the call is similar:

```
PREFIX_MODULE_TYPE (&pe, 1, len, buffer, parm)
```

the values are the same as above, except that the first parameter is a reference higher and the third and fourth parameters are a reference lower, hence –

`PREFIX_MODULE_TYPE`: indicates that the type `TYPE` in module `MODULE` should be parsed – the prefix `PREFIX` is supplied from the `ENCODER` directive (or the corresponding prefix in the `SECTIONS` directive).

`pe`: is a pointer to the presentation-element that is to be constructed.

`1`: is a magic argument.

`len`: is the integer parameter supplied for routines that encode booleans or integers, or have a length associated with them. If not present, use `NULL`.

`buffer`: is the string parameter for routines that encode string values. If not present, use `NULLCP`.

`parm`: is the user supplied parameter. This should be a variable of type `PEPYPARM` if supplied, otherwise use `NULLCP`.

These routines return the manifest constant `NOTOK` on error, or `OK` on success.

6.4.2 Diagnostic Output

When the program generated by *pepy* detects an error, it calls the routine `advise` with some diagnostic information. This routine, which should be declared `void` is defined by the user. The routine takes a variable number of arguments, similar to a *printf*(3s) routine, with the exception that the first argument is a string for *perror*(3), the second argument is the format string, and any following arguments are referenced by the format string. Consult Figure 6.11 on page 83 for an example of the `advise` procedure.

Alternately, the standard routine, `PY_advise` can be used. It simply records the information to:

6.3.2 ASN.1 Semantics

`COMPONENTS OF` and `CHOICE` types are often "pulled up" into the type which references them. This can cause anomalous behavior when things like value passing statements (e.g., `[[p expr]]`) are present in the definition of a `COMPONENTS OF` or `CHOICE` type which is pulled into another type definition.

6.4 PEPY Environment

A program generated by *pepy* expects certain things.

6.4.1 Starting Things Off

To invoke a decoding or printing routine:

```
PREFIX_MODULE_TYPE (pe, 1, len, buffer, parm)
```

where –

`PREFIX_MODULE_TYPE`: indicates that the type `TYPE` in module `MODULE` should be parsed – the prefix `PREFIX` is supplied from the `DECODER` or `PRINTER` directive (or the corresponding prefix in the `SECTIONS` directive).

`pe`: is the presentation-element that is to be examined.

`1`: is a magic argument.

`len`: is the integer parameter supplied for routines that decode values into integer parameters, or have a length associated with them. The address of an integer can be placed here if the element being parsed is a primitive type, otherwise use `NULLIP`.

`buffer`: is the string parameter for routines that decode values into string objects. The address of a character pointer can be placed here if the element being parsed is a primitive type, otherwise use `NULLVP`.

`parm`: is the user supplied parameter. This should be a variable of type `PEPYPARM` if supplied, otherwise use `NULLCP`.

```
...

BEGIN

SECTIONS build unbuild none

Salary ::=
        SEQUENCE {
            name
                PrintableString [[s parm -> name]]                    10
                %{ printf("name %s ", parm -> name); %},

            salary
                TheSalary [[i parm -> salary]]
                %{ printf("salary %d\n", parm -> salary); %}
}

TheSalary ::=
        INTEGER
                                                                      20
END
```

Figure 6.5: Examples of Parameters in Pepy (continued)

6.3 Known Deficiencies

The *pepy* program started as a prototype. Although the approach is promising, some of the rough edges remain.

6.3.1 ASN.1 Syntax

The ASN.1 syntax is followed exactly except that:

- No *macro* syntax is recognized.

SalaryDefs **DEFINITIONS** ::=

```
%{
#include <stdio.h>

char   *myname;

static struct salary_record {
    char   *name;
    int     salary;                                          10
}          salary;

#define PEPYPARM struct salary_record *


main (argc, argv)
int     argc;
char  **argv;
{
    PE        pe;                                            20

    myname = argv[0];

    salary.name = argv[1];
    salary.salary = atoi (argv[2]);
    build_SalaryDefs_Salary (&pe, 1, 0, NULLCP, &salary);

    salary.name = NULL;
    salary.salary = 0;
    unbuild_SalaryDefs_Salary (pe, 1, NULLIP, NULLVP,
```