The ISO Development Environment: User's Manual

Volume 4: The Applications Cookbook

Marshall T. Rose
Performance Systems International, Inc.

Julian P. Onions
X-Tel Services Ltd

Colin J. Robbins
X-Tel Services Ltd

February 22, 1994
(Version 7.0)

Contents

[0]This document (version #6.27) was LaTeXset February 22, 1994 with lplain v2.09-March 25, 1992.

List of Figures

Preface

The software described herein has been developed as a research tool and represents an effort to promote the use of the International Organisation for Standardisation (ISO) interpretation of open systems interconnection (OSI), particularly in the Internet and RARE research communities.

Notice, Disclaimer, and Conditions of Use

The ISODE is openly available but is **NOT** in the public domain. You are allowed and encouraged to take this software and build commercial products. However, as a condition of use, you are required to "hold harmless" all contributors.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this notice and the reference to this notice appearing in each software module be retained unaltered, and that the name of any contributors shall not be used in advertising or publicity pertaining to distribution of the software without specific written prior permission. No contributor makes any representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

All contributors disclaim all warranties with regard to this software, including all implied warranties of mechantibility and fitness. In no event shall any contributor be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in action of contract, negligence or other tortuous action, arising out of or in connection with, the use or performance of this software.

As used above, "contributor" includes, but is not limited to:

The MITRE Corporation
The Northrop Corporation
NYSERNet, Inc.
Performance Systems International, Inc.
University College London
The University of Nottingham
X-Tel Services Ltd
The Wollongong Group, Inc.
Marshall T. Rose

In particular, the Northrop Corporation provided the initial sponsorship for the ISODE and the Wollongong Group, Inc., also supported this effort. The ISODE received partial support from the U.S. Defense Advanced Research

Projects Agency and the Rome Air Development Center of the U.S. Air Force Systems Command under contract number F30602–88–C–0016 to NYSER-Net Inc.

Revision Information

This document (version #6.27) and its companion volumes are believed to accurately reflect release v 7.0 of February 22, 1994.

Release Information

If you would like a copy of the release described in this document, there are several avenues available:

- NORTH AMERICA
  For mailings in NORTH AMERICA, send a check for 375 U.S. Dollars to:

Postal address:	University of Pennsylvania Department of Computer and Information Science Moore School Attn: David J. Farber (ISODE Distribution) 200 South 33rd Street Philadelphia, PA 19104-6314 U.S.A.
Telephone:	+1 215 898 8560

  Specify one:

  1. 1600bpi 1/2–inch tape, or
  2. Sun 1/4–inch cartridge tape.

  The tape will be written in *tar* format and returned with a documentation set. Do not send tapes or envelopes. Documentation only is the same price.

- EUROPE
  For mailings in EUROPE, send a cheque or bankers draft and a purchase order for 200 Pounds Sterling to:

Postal address:	Department of Computer Science Attn: Natalie May/Dawn Bailey University College London Gower Street London, WC1E 6BT UK

For information only:

Telephone:	+44 71 380 7214
Fax:	+44 71 387 1397
Telex:	28722
Internet:	`natalie@cs.ucl.ac.uk`
	`dawn@cs.ucl.ac.uk`

Specify one:

1. 1600bpi 1/2–inch tape, or
2. Sun 1/4–inch cartridge tape.

The tape will be written in *tar* format and returned with a documentation set. Do not send tapes or envelopes. Documentation only is the same price.

- EUROPE (tape only)
  Tapes without hardcopy documentation can be obtained via the European Forum for Open Systems (EurOpen, formerly known as EUUG). The ISODE 7.0 distribution is called EurOpenD14.

Postal address:	EurOpen Software Distributions
	c/o Frank Kuiper
	Centrum voor Wiskunde en Informatica
	Kruislann 413
	1098 SJ Amsterdam
	The Netherlands
For information only:	
Telephone:	+31 20 5924056
	(or +31 20 5929333)
Telex:	12571 mactr nl
Telefax:	+31 20 5924199
Internet:	`euug-tapes@cwi.nl`

Specify one:

1. 1600bpi 1/2–inch tape: 140 Dutch Guilders.

2. Sun 1/4–inch cartridge tape (QIC-24 format): 200 Dutch Guilders.

If you require DHL this is possible and will be billed through. Note that if you are not a member of EurOpen then there is an additional handling fee of 300 Dutch Guilders (please enclose a copy of your membership or contribution payment form when ordering). Do not send money, cheques, tapes or envelopes, you will be invoiced.

- PACIFIC RIM
  For mailings in the Pacific Rim, send a cheque for 300 dollars Australian to:

Postal address:	Isode Distribution (Attn: Andrew Waugh) 723 Swanston St Carlton, VIC 3053 Australia

  For information only:

Telephone:	+61 3 282 2615
Fax:	+61 3 282 2600
Internet:	`ajw@mel.dit.csiro.au`

  Specify one:

  1. 1600/3200/6250bpi 1/2-inch tape, or
  2. Sun 1/4-inch cartridge tape in either QIC-11, QIC-24 or QIC-150 format.
  3. Exabyte 2.3 Gigabyte or 5 Gigabyte format

  The tape will be written in *tar* format and returned with a documentation set. Do not send tapes or envelopes. Documentation only is the same price.

- Internet
  If you can FTP to the Internet, you can use anonymous FTP to the host `uu.psi.com [136.161.128.3]` to retrieve `isode-7.tar.Z` in BINARY mode from the `isode/` directory. This file is the *tar* image after being run through the compress program and is approximately 4.7MB in size.

- NIFTP
  If you run NIFTP over the public X.25 or over JANET, and are registered in the NRS at Salford, you can use NIFTP with username "guest" and your own name as password, to access `UK.AC.UCL.CS` to retrieve the file `<SRC>isode-7.tar.Z`. This file is the *tar* image after being run through the compress program and is approximately 4.7MB in size.

- FTAM on the JANET, IXI or PSS
  The source code is available by FTAM at the University College London over X.25 using

  JANET (DTE `00000511160013`)

  IXI (DTE `20433450420113`)

  PSS (DTE `23421920030013`)

  Use TSEL `259` (ASCII encoding). Use the "anon" user-identity and retrieve the file `isode-7.tar.Z` from the `src/` directory. This file is the *tar* image after being run through the compress program and is approximately 4.7MB in size.

  The file service is provided by the FTAM implementation in ISODE 6.0 or later (IS FTAM) and is registered in the pilot OSI Directory below

  ```
  bells, Computer Science, University College London, GB
  ```

For distributions via either FTAM or FTP, there are two additional files available for retrieval:

- `isode-7-doc.tar.Z`
  This is the LaTeXsource for the entire documentation set. It is a compressed *tar* image (3.5MB).

- `isode-7-ps.tar.Z`
  This contains the five volume manual in PostScript format. It is a compressed *tar* image (4.2MB).

Discussion Groups

The Internet discussion group `ISODE@NIC.DDN.MIL` is used as a forum to discuss ISODE. Contact the Internet mailbox `ISODE-Request@NIC.DDN.MIL` to be asked to be added to this list.

Support

Although the ISODE is not "supported" per se, it does have a problem reporting address, `Bug-ISODE@XTEL.CO.UK`. Bug reports (and fixes) are welcome by the way.

Commercial Support

A UK company has been set up to provide commercial support for the ISODE and associated packages — X-Tel Services Ltd. This company provides an update service, general assistance and site specific support. Although inclusion of this information should not be considered an endorsement, it should be noted that two of the primary ISODE developers now work at X-Tel Services Ltd.

Postal address:	X-Tel Services Ltd. University Park, Nottingham, NG7 2RD UK
Telephone	+44 602 412648
Fax	+44 602 790278
Internet	support@xtel.co.uk

If other organisations offering formal support for the ISODE wish to be included in future announcements, a suitable index will be organized.

Acknowledgements

Many people have made comments about and contributions to the ISODE which have been most helpful. The following list is by no means complete:

The first three releases of the ISODE were developed at the Northrop Research and Technology Center, and the first version of this manual is referenced as NRTC Technical Paper #702. The initial work was supported in part by Northrop's Independent Research and Development program.

The Wollongong Group supported ISODE for its 4.0 and 5.0 release. They deserve much credit for that. Furthermore, they contributed an implementation of RFC1085, a lightweight presentation protocol for TCP/IP-based internets.

The ISODE development up until version 7.0 was supported by Performance Systems International, Inc. and NYSERNet, Inc. This was a substantial commitment to which the project in indebted. Before their support, ISODE was always an after-hours activity. The NYSERNet effort was partially supported by the U.S. Defense Advanced Research Projects Agency and the Rome Air Development Center of the U.S. Air Force Systems Command under contract number F30602–88–C–0016 to NYSERNet Inc.

Christopher W. Moore of the Wollongong Group has provided much help with ISODE in terms of both policy and implementational matters. He also performed Directory interoperability testing against a different implementation of the OSI Directory.

Dwight E. Cass of the Northrop Research and Technology Center was one of the original architects of *The ISO Development Environment.* His work was critical for the original proof of concept and should not be forgotten. John L. Romine also of the Northrop Research and Technology Center provided many fine comments concerning the presentation of the material herein. This resulted in a much more readable manuscript. Stephen H. Willson, also of the Northrop Research and Technology Center, provided some help in verifying the operation of the software on a system running the AT&T variant of UNIX[1].

The *librosap*(3n) library was heavily influenced by an earlier native-TCP version written by George Michaelson, formerly of University College London, in the United Kingdom. Stephen E. Kille, of University College London,

[1] UNIX is a trademark of AT&T Bell Laboratories.

provided valuable feedback on the *pepy*(1) utility. In addition, both Steve and George provided us with some good comments concerning the *libpsap*(3) library. Steve is also the conceptual architect for the addressing scheme used in the software, and he modified the *librosap*(3n) library to support half-duplex mode when providing ECMA ROS service. George contributed the CAMTEC and ULTRIX[2] X.25 interfaces. Simon Walton, also of University College London, has been very helpful in providing constant feedback on the ISODE during beta-testing.

The INCA project donated the QUIPU Directory implementation to the ISODE. Stephen E. Kille, Colin J. Robbins, and Alan Turland, at the time all of University College London, are the three principals who developed the 6.0 version of the directory software. In addition, Steve Titcombe, also of UCL spent considerable time on the DIrectory SHell (DISH), and Mike Roe formerly of UCL, put a large amount of effort into the security requirements of QUIPU. Development of the current version of QUIPU has been coordinated by Colin J. Robbins now of X-Tel Services Ltd, and designed by Stephen E. Kille.

The UCL work has been partially supported by the commission of the EEC under its ESPRIT program, as a stage in the promotion of OSI standards. Their support has been vital to the UCL activity. In addition, QUIPU is also funded by the UK Joint Network Team (JNT).

Julian P. Onions, of X-Tel Services Ltd is the current *pepy*(1) guru, having brainstormed and implemented the encoding functionality along with Stephen E. Easterbrook formerly of University College London. Julian also contributed the UBC X.25 interface along with the TCP/X.25 TP0 bridge, and has also contributed greatly to *posy*(1). Julian's latest contribution has been a *transport service bridge*. This is used to masterfully solve interworking problems between different OSI stacks (TP0/X.25, TP4/CLNP, RFC1006/TCP, and so on).

John Pavel and Godfrey Cowin of the Department of Trade and Industry/ National Physical Laboratory in the United Kingdom both contributed significant comments during beta-testing. In particular, John gave us a lot of feedback on *pepy*(1) and on the early FTAM DIS implementation. John also contributed the SunLink X.25 interface.

Keith Ruttle of CAMTEC Electronics Limited in the United Kingdom

[2] ULTRIX is a trademark of Digital Equipment Corporation.

contributed both the driver for the new CAMTEC X.25 interface and the CAMTEC CONS interface (X.25 over 802 networks). This latter driver was later removed from the distribution for lack of use.

In addition, Andrew Worsley of the Department of Computer Science at the University of Melbourne in Australia pointed out several problems with the FTAM DIS implementation. He also developed a replacement for *pepy* and *posy* called *pepsy*. After moving to University College London, he improved this system and integrated into the ISODE.

Olivier Dubous of BIM sa in Belgium contributed some fixes to concurrency control in the FTAM initiator to allow better interworking with the VMS[3] implementation of the filestore. He also suggested some changes to allow interworking with the FTAM T1 and A/111 profiles.

Olli Finni of Nokia Telecommunications provided several fixes found when testing interoperability with the TOSI implementation of FTAM.

Mark R. Horton of AT&T Bell Laboratories also provided some help in verifying the operation of the software on a 3B2 system running UNIX System V release 2. In addition, Greg Lavender of NetWorks One, under contract to the U.S. Navy Regional Data Automation Center (NARDAC), provided modifications to allow the software to run on a generic port of UNIX System V release 3.

Steve D. Miller of the University of Maryland provided several fixes to make the software run better on the ULTRIX variant of UNIX.

Jem Taylor of the Computer Science Department at the University of Glasgow provided some comments on the documentation.

Hans-Werner Braun of the University of Michigan provided the inspiration for the initial part of Section 1.2.

A previous release of the software contained an ISO TP4/CLNP package derived from a public-domain implementation developed by the National Institute of Standards and Technology (then called the National Bureau of Standards). The purpose of including the NIST package (and associated support) was to give an example of how one would interface the code to a "generic" TP4 implementation. As the software has now been interfaced to various native TP4 implementations, the NIST package is no longer present in the distribution.

John A. Scott of the MITRE Corporation contributed the SunLink OSI

[3] VMS is a trademark of Digital Equipment Corporation.

interface for TP4. He also wrote the FTAM/FTP gateway which the MITRE Corporation has generously donated to this package.

Philip B. Jelfs of the Wollongong Group upgraded the FTAM/FTP gateway to the "IS-level" (International Standard) FTAM.

Rick Wilder and Don Chirieleison of the MITRE Corporation contributed the VT implementation which the MITRE Corporation has generously donated to this package.

Jacob Rekhter of the T. J. Watson Research Center, IBM Corporation, provided some suggestions as to how the system should be ported to the IBM RT/PC running either AIX or 4.3BSD. He also fixed the incompatibilities of the FTAM/FTP gateway when running on 4.3BSD systems.

Ashar Aziz and Peter Vanderbilt, both of of Sun Microsystems Inc., provided some very useful information on modifying the SunLink OSI interface for TP4.

Later on, elements of the SunNet 7.0 Development Team (Hemma Prafullchandra, Raj Srinivasan, Daniel Weller, and Erik Nordmark) made numerous enhancements and fixes to the system.

John Brezak of Apollo Computer, Incorporated ported the ISODE to the Apollo workstation. Don Preuss, also of Apollo, contributed several enhancements and minor fixes.

Ole-Jorgen Jacobsen of Advanced Computing Environments provided some suggestions on the presentation of the material herein.

Nandor Horvath of the Computer and Automation Institute of the Hungarian Academy of Sciences while a guest researcher at the DFN/GMD in Darmstadt, FRG, provided several fixes to the FTAM implementation and documentation.

George Pavlou and Graham Knight of University College London contributed some management instrumentation to the *libtsap*(3n) library.

Juha Heinänen of Tampere University of Technology provided many valuable comments and fixes on the ISODE.

Paul Keogh of the Nixdorf Research and Development Center, in Dublin, Ireland, provided some fixes to the FTAM implementation.

Oliver Wenzel of GMD Berlin contributed the RFA system.

L. McLoughlin of Imperial College contributed Kerberos support for the FTAM responder.

Kevin E. Jordan of CDC provided many enhancements to the G3FAX library.

John A. Reinart of Cray Research contributed many performance enhancements.

Ed Pring of the T. J. Watson Research Center, IBM Corporation, provided several fixes to the SMUX implementation in ISODE's SNMP agent.

Uli Betzler of University of Karlsruhe donated the HP X.25 interface and other HP-UX modifications.

Alan Young of Concurrent Computer Corporation contributed many helpful changes to the transport code.

James Gosling, author of the superb Emacs screen editor for UNIX, and Leslie Lamport, author of the excellent LaTeX document preparation system both deserve much praise for such winning software. Of course, the whole crew at U.C. Berkeley also deserves tremendous praise for their wonderful work on their variant of UNIX.

Finally, full credit for this release must be given to Marshall T. Rose. Marshall has for so long master-minded the whole ISODE effort. Although he has not been involved in the last month or so leading upto this release, the vast majority of the effort getting ISODE-7.0 into shape came from Marshall.

jpo & cjr

Nottingham, England
February, 1994

Part I

Introduction

Chapter 1

Overview

This document describes a non-proprietary implementation of some of the protocols defined by the International Organisation for Standardisation and International Electrotechnical Commission (ISO/IEC), the International Telegraph and Telephone Consultative Committee (CCITT), and the European Computer Manufacturer Association (ECMA).[1]

The purpose of making this software openly available is to accelerate the process of the development of applications in the OSI protocol suite. Experience indicates that the development of application level protocols takes as long as or significantly longer than the lower level protocols. By producing a non-proprietary implementation of the OSI protocol stack, it is hoped that researchers in the academic, public, and commercial arenas may begin working on applications immediately. Another motivation for this work is to foster the development of OSI protocols both in the European RARE and the U.S. Internet communities. The Internet community is widely known as having pioneered computer-communications since the early 1970's. This community is rich in knowledge in the field, but currently is not actively experimenting with the OSI protocols. By producing an openly available implementation, it is hoped that the OSI protocols will become quickly widespread in the Internet, and that a productive (and *painless*) transition in the defense data network (DDN) might be promoted. The RARE community is the set of corresponding European academic and research organisations. While they do not have the same long implementation experience as the Internet

[1] In the interests of brevity, unless otherwise noted, the term "OSI" is used to denote these parallel protocol suites.

community, they have a deep commitment to international standards. It is intended that this release gives vital early access to prototype facilities.

1.1 Fanatics Need Not Read Further

This software can support several different network services below the transport service access point (TSAP). One of these network services is the DoD transmission control protocol (TCP)[Pos81].[2] This permits the development of the higher level protocols in a robust and mature internet environment, while providing us the luxury of not having to recode anything when moving to a network where the OSI transport protocol (TP) is used to provide the TSAP. However, the software also operates over pure OSI lower levels of software. It is mainly used in that fashion — outside of the United States.

Of course, there will always be "zealots of the pure faith" making claims to the effect that:

> *TCP/IP is dead! Any work involving TCP/IP simply dilutes the momentum of OSI.*

or, from the opposite end of the spectrum, that

> *The OSI protocols will never work!*

Both of these statements, from diametrically opposing protocol camps are, of course, completely unfounded and largely inflammatory. TCP/IP is here, works well, and enjoys a tremendous base of support. OSI is coming, and will work well, and when it eventually comes of age, it will enjoy an even larger base of support.

The role of ISODE, in this maelstrom that generates much heat and little light, is to provide a useful transition path between the two protocol suites in which complementary efforts can occur. The ISODE approach is to use the strengths of both the DDN and OSI protocol suites in a cooperative and positive manner. For a more detailed exposition of these ideas, kindly refer to [Ros90] or the earlier work [RC86].

---

[2] Although the TCP corresponds most closely to offering a transport service in the OSI model, the TCP is used as a connection-oriented network protocol (i.e., as coservice to X.25).

1.2 The Name of the Game

The name of the software is the ISODE. The official pronunciation of the ISODE, takes four syllables: *I-SO-D-E*. This choice is mandated by fiat, not by usage, in order to avoid undue confusion.

Please, as a courtesy, do not spell ISODE any other way. For example, terms such as ISO/DE or ISO-DE do not refer to the software! Similarly, do not try to spell out ISODE in such a way as to imply an affiliation with the International Organisation for Standardization. There is no such relationship. The *ISO* in ISODE is not an acronym for this organisation. In fact, the *ISO* in ISODE doesn't really meaning anything at all. It just has a catchy two syllable sound.

1.3 Operating Environments

This release is coded entirely in the *C* programming language[KR78], and is known to run under the following operating systems (without any kernel modifications):

Berkeley UNIX The software should run on any faithful port of 4.2BSD, 4.3BSD, or 4.4BSD UNIX. Sites have reported the software running: on the Sun-3 workstation running Sun UNIX 4.2 release 3.2 and later; on the Sun Microsystems workstation (Sun-3, Sun-4, and Sun-386i) running SunOS release 4.0 and later; on Digital Equipment Corporation ULTRIX machines (including X.25); the Integrated Solutions workstation; and on the RT/PC running 4.3BSD.[3] In addition to using the native TCP facilities of Berkeley UNIX, the software has also be interfaced to versions 4.0 through 7.0 of the SunLink/SunNet X.25 and OSI packages (although Sun may have to supply you with some modified `sgtty` and `ioctl` include files if you are using an earlier version of SunLink X.25). The optional SunLink Communications Processor running

[3] Do not however, attempt to compile the software with the SunPro *make* program! It is not, contrary to any claims, compatible with the standard *make* facility. Furthermore, note that if you are running a version of SunOS 4.0 prior to release 4.0.3, then you may need to use the *make* program found in */usr/old/*, if the standard *make* you are using is the SunPro *make*. In this case, you will need to put the old, standard *make* in */usr/bin/*, and you can keep the SunPro *make* in */bin/*.

DCP 3.0 software has also been tested with the software.

AT&T UNIX The software should run on any faithful port of System V UNIX. The SVR4 port used an ICL DRS6000 as a testbed. One of the other systems tested was running with an Excelan EXOS[4] 8044 TCP/IP card. The Excelan package implements the networking semantics of the 4.1aBSD UNIX kernel. As a consequence, the software should run on any faithful port of 4.1aBSD UNIX, with only a minor amount of difficultly. As of this writing, however, this speculation has not been verified. The particular system used was a Silicon Graphics IRIS workstation.[5]

Another system was running the WIN TCP/IP networking package. The WIN package implements the networking semantics of the 4.2BSD UNIX kernel. The particular system used was a 3B2 running System V release 2.0.4, with WIN/3B2 version 1.0.

Another system was also running the WIN TCP/IP networking package but under System V release 3.0. The WIN package on SVR3 systems emulates the networking semantics of the 4.2BSD UNIX kernel but uses STREAMS and TLI to do so.

Note: The System V port does not include VT or SNMP.

AIX The software should run on the IBM AIX Operating System which is a UNIX-based derivative of AT&T's System V. The particular system used was a RT/PC system running version 2.1.2 of AIX.

HP-UX The software should run on HP's UNIX-like operating system, HP-UX. The particular system used was an Indigo 9000/840 system running version A.B1.01 of HP-UX. The system was also reported to have run on an HP 9000/350 system under version 6.2 of HP-UX. Support for HP's X.25 has been included.

ROS The software should run on the Ridge Operating system, ROS. The particular system used was a Ridge-32 running version 3.4.1 of ROS.

[4] EXOS is a trademark of Excelan, Incorporated.

[5] This test was made with an earlier release of this software, and access to an SGI workstation was not available when the current version of the software tested. However, the networking interface is still believed to be correct for the Excelan package.

Pyramid OsX The software should run on a Pyramid computer running OsX. The particular system used was a Pyramid 98xe running version 4.0 of OsX.

Since a Berkeley UNIX system is the primary development platform for ISODE, this documentation is somewhat slanted toward that environment.

1.4 Organisation of the Release

A strict layering approach has been taken in the organisation of the release. The documentation mimics this relationship approximately: the first two volumes describe, in top-down fashion, the services available at each layer along with the databases used by those services; the third volume describes some applications built using these facilities; the fourth volume describes a facility for building applications based on a programming language, rather that network-based, model; and, the fifth volume describes a complete implementation of the OSI directory.

In *Volume One*, the "raw" facilities available to applications are described, namely four libraries:

- The *libacsap*(3n) library, which implements the OSI association control service (ACS).
- The *librosap*(3n) library, which implements different styles of the OSI remote operations service (ROS).
- The *librtsap*(3n) library, which implements the OSI reliable transfer service (RTS). and,
- The *libpsap*(3) library, which implements the OSI abstract syntax and transfer mechanisms.

In *Volume Two*, the services upon which the application facilities are built are described, namely three libraries:

- The *libpsap2*(3n) library, which implements the OSI presentation service.
- The *libssap*(3n) library, which implements the OSI session service. and,

- The *libtsap*(3n) library, which implements an OSI transport service access point.

In addition, there is a replacement for the *libpsap2*(3n) library called the *libpsap2-lpp*(3n) library. This implements the lightweight presentation protocol for TCP/IP-based internets as specified in RFC-1085.

In addition, *Volume Two* contains information on how to configure the ISODE for your network.

In *Volume Three*, some application programs written using this release are described, including:

- An implementation of the ISO FTAM which runs on Berkeley or AT&T UNIX. FTAM (file transfer, access and management), is the OSI file service. The implementation provided is fairly complete in the context of the particular file services it offers. It is a minimal implementation inasmuch as it offers only four core services: transfer of text files, transfer of binary files, directory listings, and file management.

- An implementation of an FTAM/FTP gateway, which runs on Berkeley UNIX.

- An implementation of the ISO VT which runs on Berkeley UNIX. VT (Virtual Terminal) is the OSI terminal service. The implementation consists of a basic class, TELNET profile implementation.

- An implementation of the "little services" often used for debugging and amusement.

- An implementation of a simple image database service.

In *Volume Four*, a "cooked" interface for applications using remote operations is described, which consists of three programs and a library:

- The *rosy*(1) compiler, which is a stub-generator for specifications of remote operations.

- The *posy*(1) compiler, which is a structure-generator for ASN.1 specifications.

- The *pepy*(1) compiler, which reads a specification for an application and produces a program fragment that builds or recognises the data structures (APDUs in OSI argot) which are communicated by that application.

- The new *pepsy*(1) compiler which will replace *pepy* and *posy* programs in a future release of the ISODE.

- The *librosy*(3n) library, which is a library for applications using this distributed applications paradigm.

In *Volume Five*, the QUIPU directory is described, which currently consists of several programs and a library:

- The *quipu*(8c) program, which is a directory system agent (DSA);

- The *dish*(1c) family of programs, which are a set of directory shell commands; and,

- The *libdsap*(3n) library, which is a library for applications using the directory.

1.5 A Note on This Implementation

Although the implementation described herein might form the basis for a production environment in the near future, this release is not represented as "production software".

However, throughout the development of the software, every effort has been made to employ good software practices which result in efficient code. In particular, the current implementation avoids excessive copying of bytes as data moves between layers. Some rough initial timings of echo and sink entities at the transport and session layers indicate data transfer rates quite competitive with a raw TCP socket (most differences were lost in the noise). The work involved to achieve this efficiency was not demanding.

Additional work was required so that programs utilising the *libpsap*(3) library could enjoy this level of performance. Although data transfer rates at the reliable transfer and remote operations layers are not as good as raw TCP, they are still quite impressive (on the average, the use of a ROS interface (over presentation, session, and ultimately the TCP) is only 20% slower than a raw TCP interface).

1.6 Changes Since the Last Release

A brief summary of the major changes between v. 6.0 and v. 7.0 are now presented. These are the user-visible changes only; changes of a strictly internal nature are not discussed.

- A new program, *pepsy*, has been developed to replace both *pepy* and *posy*. It is described in *Volume Four*.
- The *dsabuild* program has been removed, in favor of some shell scripts.
- The "higher performance nameservice" has been discontinued in favor of a "user-friendly nameservice". As such, the syntax of the `str2aei` routine has changed. This routine will soon be deprecated, so get in the habit of using the new `str2aeinfo` routine discussed in the ISODE Manual.
- The `na_type` and `na_subnet` fields of the network address structure described in the ISODE Manual have been renamed. For compatibility, macros are provided. These macros will be removed after this release.
- The stub directory facility is now deprecated in favor of an OSI directory based approach. As a result, the *aetbuild* program has been removed.
- An interworking problem in the presentation layer has been fixed in a backwards compatible manner. This should not affect interoperability in this release. The backwards compatibility will be removed after this release.

As a rule, the upgrade procedure is a two-step process: first, attempt to compile your code, keeping in mind the changes summary relevant to the code; and, second, once the code successfully compiles, run the code through *lint*(1) with the supplied lint libraries.

Although every attempt has been made to avoid making changes which would affect previously coded applications, in some cases incompatible changes were required in order to achieve a better overall structure.

Part II

Overview

Chapter 2

Motivation and Concepts

Remote operations are a well understood technique for building distributed applications. Currently, OSI technology provides a powerful notation for specifying the external interactions of these systems. Remote operations in OSI are particularly attractive as they specify the homogeneous externally visible and verifiable characteristics needed for interconnection compatibility, while avoiding unnecessary constraints upon and changes to the heterogeneous internal design and the implementation of the information processing systems to be interconnected. Many feel that this capability is likely to be a key factor in the overall success of OSI standardization.

In [ECM85], a methodology for remote operations is presented which aims to package this technology concisely. Although the latter half of this document has been rendered obsolete by more recent work in the standards bodies (e.g., [ISO88a, ISO88b]) The aim of this volume, *The Applications Cookbook*, is to provide programming support for this methodology.

2.1 A Model for Distributed Applications

Remote operations are viewed as part of a methodology for building distributed applications.[1] In order to understand the role remote operations play in this programming model, one other notion must be explained first.

[1]This section presents a (hopefully) simplified version of the information presented in the first half of [ECM85]. Interested readers should consult this original work for further details.

2.1.1 Abstract Data Types

An *abstract data type* is a concept for describing a data structure which is accessed in a well-defined manner. This has several implications:

- Although the data structure may have a *concrete representation* on a given local system (e.g., a *C* `struct`), its corresponding abstract data type is defined in an implementation-independent fashion, termed its *abstract syntax*.

- A well-defined set of rules, defined by an *abstract transfer notation*, is used to serialize the abstract syntax so that a data structure corresponding to the abstract data type may be unambiguously transmitted on the network. There are actually two mappings here:

  - First, the data structure is mapped to the abstract syntax for the abstract data type. This is an implementation-dependent mapping.
  - Second, the abstract syntax is mapped to the concrete syntax. This is a serializing activity resulting in a stream of octets (although some concrete syntaxes produce bit-aligned, rather than octet-aligned, streams). Do not confuse the concrete representation described above (which deals with programming languages), with the concrete syntax (which deals with transmission properties).

- Access to an abstract data type is defined by a set of unitary actions termed *operations*, which define the complete behavior of an instance of the abstract data type.

- *Strong typing* results when operations are the only permitted means of accessing an abstract data type.

- An *object model* for programming follows from the use of abstract data types rather than concrete data structures. Since operations are used to ultimately manipulate the data structures, an important level of indirection has been introduced: data structures may be accessed without regard to their actual implementation.

2.1.2 Operations

Having described the relationship between data structures, abstract data types, and operations, consider the generic structure of an operation. An *operation*, in its most primitive form, is a simple request/reply interaction. The interaction takes the following form:

- The operation is *invoked*. An invocation consists of:
  - an *operation number*, which uniquely identifies the operation to be performed;
  - an arbitrarily complex *argument*;
  - an *invocation identifier*, which is used to distinguish this invocation from other previous invocations; and,
  - possibly a *linked invocation identifier*, which is used to indicate that this invocation results during the processing of a previous invocation.
- If the operation succeeds, then a *result* may be returned. A result consists of:
  - an invocation identifier corresponding to the operation which succeeded; and,
  - (possibly) an arbitrarily complex *result*.
- If the operation fails, then an *error* is returned. An error consists of:
  - an invocation identifier corresponding to the operation which failed;
  - an *error number*, which uniquely identifies the error that occurred; and,
  - (possibly) an arbitrarily complex *parameter*.
- If the operation was not performed for some reason (e.g., an unknown operation number, mistyped arguments for the operation, and so on), then a *reject* is returned. A reject consists of:
  - an invocation identifier corresponding to the operation which was performed (in degenerate cases this information may not be available); and,

– a *reason*, which describes, in general terms, the rejection which occurred.

Note that in all cases, operations are seen as being *total*. This means that for any given operation, the normal outcome (the result) and all exception outcomes (the errors) are well-defined and mutually exclusive (unambiguously distinguished).

2.1.3 Associations

Thus far, we have implicitly assumed some communications relationship (i.e., a *binding*) between the entity invoking operations, the *invoker*, and the entity performing them, *performer*. It is necessary formalize these notions somewhat.

An *association* is a binding between two entities, which are referred to as the *initiator* and the *responder*. Although this sounds straight-forward, there is actually one level of indirection present: the initiator does not directly select a responder for binding, but rather selects a *service* that it wishes to use. During this process a responder is located and the binding established.

The binding process is actually two-step: first, the initiator determines which service it requires and asks to have this service mapped onto the entities available on the network; second, based on the initiator's communications requirements, an association will be bound to one of those entities, which becomes the responder.

Note that there is no requirement that the initiator of an association be the entity which requests operations to be invoked (i.e., the initiator need not be an invoker), nor that the responder of an association be the entity which performs the operations (i.e., the responder need not be a performer).

2.2 Design Guidelines

Although the characteristics of operations will vary between applications, there are certain guidelines which should be of universal interest.

2.2.1 Reliability Characteristics

When executing an operation in a distributed environment, there is always an element of uncertainity with regards to the occurrence of remote events, particularly when failures occur. One classification of reliability might be as follows: for a given invocation, a guarantee is given that it occurs *exactly once*, *at least once*, or *at most once*.

The first two guarantees require an end-to-end confirmation, with the proviso that the semantics of *exactly once* require a recovery scheme in the event of failures. When an operation has the semantics of *at least once*, it is called *idempotent*. These are operations which read information, but do not modify the remote state.

In practice, implementing these semantics can be surprisingly straightforward, given the judicious use of invocation identifiers:

exactly once: The invoker repeatedly requests the operation (using the same invocation identifier) until either a confirmation (result or error) is received or a rejection of "duplicate operation" is received. The performer keeps track of the invocation identifiers of all performed operations requested by an entity from an epoch date. If the performer finds an invocation identifier being repeated, rather than perform the operation, it rejects the operation as a "duplicate operation".

at least once: The invoker repeatedly requests the operation (with any invocation identifier) until a confirmation (result or error) is received. The performer has no state information regarding previously used invocation identifiers.

at most once: The invoker requests the operation exactly once with any invocation identifier. The performer has no state information regarding previously used invocation identifiers.

2.2.2 Keeping Total Operations Total

An operation may be thought of as *confirmed* when either a result or error is returned by the performer. However, under the OSI framework for remote operations, it is possible to have an operation which may only return a result, or may only return errors. This is a subtle point, but one worth emphasizing: in the past some operations have been defined which on success do not return a result, this lead to ambiguity in as much as the invoker could not determine the "correct" time to terminate the association, since "silence" on the part of the performer could mean that the operation completed successfully or that the operation was still in progress and an error return might be forthcoming. In keeping with the philosophy of total operations, it is important to have an operation return a result, even if the information is `NULL`.

2.3 For Further Reading

The first half of [ECM85] is the most authoritative reference on using remote operations. In particular, Section 4.4 presents a simple template methodology for using remote operations. As mentioned earlier, Section 5 and beyond have been obsoleted by work in ISO and CCITT.

In as much as the standards from these bodies deal with the remote operations service and protocol, they are not particularly germane here. The ISO work on remote operations is presented in [ISO88a] and [ISO88b]. The corresponding CCITT work is described in [CCI88a] and [CCI88b]. The native interface to these services are discussed in Chapter 3 of *Volume One*.

Chapter 3

A Discipline for Meal Preparation

The previous chapter presented a methodology for building distributed applications, based on two key aspects: *abstract data types*, and *remote operations*. During the exposition of these abstract terms, they were related to their more concrete realization in an OSI architecture, namely: abstract syntax, association control, and remote operations.

In the chapters that follow, this volume will introduce the utensils (programming tools) and various recipes (boilerplate routines) which can be used to cook-up OSI applications. Future editions of this volume will no doubt detail how one can prepare either fast food, nouvelle cuisine, or gourmet cooking. Given the raw power of remote operations in OSI, the cooking analogy is actually quite appropriate: some features of the remote operations service will be restricted in order to make the programming work more digestable!

In this chapter, the steps described in the following chapters are introduced and placed in perspective. Throughout the remaining chapters, an extended example is presented to illustrate, step-by-step, the use of remote operations.

3.1 Defining A New Service

A collection of remote operations forms *only* a partial definition of a service. Other parts of this definition include naming and addressing information.

Chapter 10 of *Volume One* describes fully how a service is defined in these terms. Rather than reproduce that chapter in this volume, let us consider the essential points. At first reading, this seems complicated. In point of fact, the first definition of a service is challenging, defining later services is straight-forward and only marginally tedious!

Here are the ones we are interested in:

abstract syntax: this describes the data structures being exchanged by the service;

application context name: this describes the protocol being used by the service;

application-entity information: this uniquely names an entity in the network;

presentation address: this locates an entity in the network; and,

local program: this identifies the program on the local system which implements the service.

We now begin the example which will be repeatedly extended in the chapters which follow. Let us suppose that we are defining the *ISODE Image Service*, which offers a simple image database service.

Although Chapter 10 of *Volume One* defines several approaches, we will take the "standard" approach for our example:

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 image 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 image"         1.17.2.7.2
```

application-entity information/presentation address: defined with:

```
default servicestore    1.17.4.1.n  ""  ""  #p
```

in the *isoentities*(5) file (we say that `servicestore` is the "qualifier" portion of the service). If we select `p` as the lowest unassigned TSAP ID between `1024` and `2047` inclusive, e.g., `1040`, we might have:

```
default imagestore     1.17.4.1.7  ""  ""  #1040
```

local program: defined in the *isoservices*(5) file as:

```
"tsap/servicestore"    #p  program arg1 arg2 ... argN
```

If we select *ros.image* as the UNIX program implementing the *ISODE Image Service*, we might have something like:

```
"tsap/imagestore"    #1040  ros.image
```

By convention, the names of responder programs start with `"ros."`.

In future releases of the software, a facility might be added to determine most of this information directly from the formal definition of the service (i.e., from the remote operations module described momentarily).

The definition of this naming and addressing information is an administrative necessity. With this out of the way, let's consider the actual work that goes into building a distributed application which uses remote operations.

3.2 Defining A Remote Operations Module

A *remote operations module* defines the operations (and associated errors) along with the abstract syntax of the data structures exchanged by the service. The *rosy*(1) program reads a description of the module and produces the corresponding *C* stubs and definitions used in accessing the service.

In simple terms, what this means is that *rosy* will produce routines that the invoker calls that will request the operation by the performer.

3.3 Defining Concrete Data Structures

An *abstract syntax module* defines the data structures being exchanged by the service. In many senses, it is equivalent to a remote operations module with the definitions of the operations removed. The *posy*(1) program reads a description of the module and produces the corresponding *C* structure definitions along with an augmented module that is read by the *pepy* compiler. The *C* structure definitions are used by the invoker and the performer, the run-time environment is responsible for mapping these data structures to the abstract syntax and then for mapping the abstract syntax to the concrete syntax.

In simple terms, what this means is that the invoker and performer worry about only the native machine *C* structures, and the open systems interface is handled entirely automatically.

3.4 Building An Initiator

In general, an initiator might take one of two implementational forms:

interactive: the user runs a program and interactively directs the invocation of operations; or,

embedded: as a part of its running, the program automatically forms an association and invokes operations as required.

The choice of form is up to the implementor. Note that in both cases, the initiator is also acting as the invoker.

In order to speed development, complete boilerplate for an interactive initiator, along with partial boilerplate for an embedded initiator is provided.

3.5 Building A Responder

In general, a responder may also take on of two implementational forms:

single association: each time the service is requested, a new instantiation of the program implementing the service is executed (this is termed the *dynamic* approach); or,

multiple association: each time the service is requested, the request is given to a single, already executing, instantiation of the program which implements the service (this is termed the *static* approach).

Although Section 3.1 describes how to define addressing information for the dynamic approach, it is good practice to implement each responder to be capable of operating in either mode and then leaving the decision to the system administrator.

In order to speed development, complete boilerplate for a dual-approach responder is provided.

3.6 Putting It All Together

In addition to the skeletal information above, a set of consistent policies for file names, rules for *make*(1), and so on, are needed. These will be introduced at the appropriate time.

It is important to understand that this volume defines **one** possible discipline for using remote operations. There are, of course, **many** possible disciplines for accomplishing the same goal. (And some will most likely be vastly superior.) The particular discipline described in *The Applications Cookbook* was chosen primarily for its balance of relative simplicity and extensibility. For some developers, it will still be too under done; if this is the case, you are encouraged to build further cooked interfaces on top of these facilities.

Part III

Utensils

Chapter 4

Remote Operations Stub-Generator

The *rosy*(1) program is a compiler for specifications of remote operations.

For those interested in trivia, *rosy* stands for Remote Operations Stub-generator (YACC-based). The *yacc*(1) program, of course, is a compiler-compiler.

4.1 Remote Operations Module

The *rosy* program reads a description of a *remote operations module* and produces the corresponding *C* stubs and definitions for use with the run-time environment provided by the *librosy*(3n) library. The syntax of the input file corresponds almost exactly to the remote operations notation described in any of the references found in Section 3.7 on page 84 of *Volume One*. (But see Section 4.3 on page 36 for the details.) In short, three kinds of objects are defined: *operations*, *errors*, and *types*.

An example of a complete remote operations module can be found in Figure 4.1 starting on page 34. For now, let's briefly consider the three individual objects.

4.1.1 Operation Definitions

Operations are defined by a name which starts with a lowercase character followed by the keyword `OPERATION`. Following this are optional definitions for the operation:

- `ARGUMENT`: defines the ASN.1 type which the operation expects as its argument;
- `RESULTS`: defines the ASN.1 type which the operation returns on success; and,
- `ERRORS`: defines the errors which the operation can return on failure.

These optional definitions must occur in the order presented here. Following any optional definitions, the lexeme "`::=`" follows with an integer which uniquely identifies the operation in the module. By convention, the integer, termed the *operation code*, should be non-negative.

4.1.2 Error Definitions

Errors are defined by a name which starts with a lowercase character followed by the keyword `ERROR`. Following this are optional definitions for the operation:

- `PARAMETER`: defines the ASN.1 type which the error returns when it occurs.

Following any optional definitions, the lexeme "`::=`" follows with an integer which uniquely identifies the error in the module. By convention, the integer, termed the *error code*, should be non-negative.

4.1.3 Type Definitions

Types are defined by a name which starts with an uppercase character followed by the lexeme "`::=`" and then an ASN.1 type definition. As the ASN.1 language is rich in its notation, the rules are not presented herein. The reader should consult [ISO87, CCI88c] for the full details, though Section 6.2.1 on page 60 of *Volume One* presents an incomplete treatment of the less substantive aspects of the language.

```
IMAGE DEFINITIONS ::=

BEGIN

-- operations

initialize OPERATION
        ARGUMENT   Initialize-INV
        RESULT     Initialize-RSP
        ERRORS     { congested, versionNotSupported }
        ::=        1

readImage OPERATION
        ARGUMENT   ReadImage-INV
        RESULT     ReadImage-RSP
        ERRORS     { congested, noMatches, formatNotSupported }
        ::=        2


-- errors

congested ERROR
        ::=        1

versionNotSupported ERROR
        PARAMETER  VersionsSupported
        ::=        2

noMatches ERROR
        ::=        3

formatNotSupported ERROR
        PARAMETER  FormatsSupported
        ::=        4


-- types
```

```
Initialize−INV ::=
        SEQUENCE {
           proposed[0]
                 IMPLICIT Version
                 DEFAULT { version−1 }
        }

Initialize−RSP ::=
        SEQUENCE {
           negotiated[0]            −− need not be present if only one
                                    −− version was requested and that was chosen,
                                    −− otherwise exactly one bit is set
                 IMPLICIT Version
                 OPTIONAL
        }

VersionsSupported ::=
        SEQUENCE {
           supported[0]
                 IMPLICIT Version
        }

Version ::=
        BITSTRING { version−1(0) }


ReadImage−INV ::=
        SEQUENCE {
                                    −− in order of preference for searching
           SEQUENCE OF
                 CHOICE {
                    file−name[0]
                          IMPLICIT Filename,

                    mail−address[1]
                          IMPLICIT Mail−Address
                 },

                                    −− in order of preference for retrieval
```

```
        SEQUENCE OF
            Format
    }

ReadImage-RSP ::=
    SEQUENCE {
        CHOICE {
            x-windows
                X-Windows-Bitmap
        }
    }

FormatsSupported ::=
    SEQUENCE {
        supported[0]
            IMPLICIT Format
    }

Format ::=
    BITSTRING { x-windows-bitmap(0) }


-- discriminants

Filename ::=
    [APPLICATION 1]
        IMPLICIT GraphicString

Mail-Address ::=
    [APPLICATION 2]
        IMPLICIT SEQUENCE {
            local[0]
                IMPLICIT GraphicString,

            domain[1]
                IMPLICIT GraphicString,

            options[2]
                IMPLICIT BITSTRING { default-local(0), default-host(1) }
```

```
                DEFAULT { default−local, default−host }
        }

−− image formats

X−Windows−Bitmap ::=
        [APPLICATION 100]
           IMPLICIT SET {
                width[0]
                   IMPLICIT INTEGER,

                height[1]
                   IMPLICIT INTEGER,

                hot−spot−x[2]
                   IMPLICIT INTEGER
                   DEFAULT −1,

                hot−spot−y[3]
                   IMPLICIT INTEGER
                   DEFAULT −1,

                data[4]             −− padded to short boundaries
                   IMPLICIT OCTETSTRING
        }

END
```

Figure 4.1: Example of a Remote Operations Module

4.2 ROSY Environment

The *rosy* program will produce several files after reading its remote operations module.

4.2.1 Abstract Syntax Module

The abstract syntax module produced by *rosy* is simply a copy of the type definitions found in the remote operations module. In addition, *rosy* may define a few new types based on the definitions used in the operations and errors.

4.2.2 C Language Stubs

Three separate stub files are produced by *rosy*.

The first is a *C* pre-processor file containing definitions of:

- each operation code

  ```
  #define operation_MODULE_OPNAME OPCODE
  ```

  as in:

  ```
  #define operation_IMAGE_initialize 1
  ```

- each error code

  ```
  #define error_MODULE_ERNAME ERRCODE
  ```

  as in:

  ```
  #define error_IMAGE_congested 1
  ```

- stub routines

  ```
  #define op_MODULE_OPNAME(sd,in,out,rsp,roi) \
                                  RyOperation ( ... )
  ```

  as in:

  ```
  #define op_IMAGE_initialize(sd,in,out,rsp,roi) \
                                  RyOperation ( ... )
  ```

The stub routines require a lot more explanation. Rather than introduce the full details, for the moment simply note that the stub routines are actually calls to the `RyOperation` routine discussed in Chapter 8 which "does all the work".

The second file produced by *rosy* defines the *C* language data structures used by the run-time environment which the *librosy*(3n) library provides.

The third file produced by *rosy* is used in conjunction with *lint*(1). It defines as routines, rather than macros, the same stub routines found in the *C* pre-processor file. These routines permit more stringent argument checking.

4.3 Known Deficiences

The *rosy* program is a prototype. It probably won't undergo substantive change, but experience will tell.

In particular, *rosy* uses essentially the same front-end as the *pepy*(1) program, so it has some limitations in the ASN.1 syntax it can accept. Consult Section 6.3.1 for the details.

The `BIND` and `UNBIND` macros are currently unrecognized, along with the newer (`OBJECT IDENTIFIER`) notation for operation codes.

The `LINKED` clause is currently recognized but otherwise ignored for the `OPERATION` macro.

4.4 Running ROSY

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

4.4.1 Options

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

The '`-b`' switch sets the prefix used for the output file. If this option is not given, the ASN.1 module name is used.

The '`-d`' switch directs *rosy* to ignore any *pepy*-style augmentations.

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.

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

The '`-pepsy`' switch tells *rosy* that you will be using *pepsy* rather than *posy*. The *pepsy* program is dscribed in Chapter 7 starting on page 88.

4.4.2 Makefiles

By convention, input files to *rosy* have the extension `.ry`.

Consider the file `module.ry` which defines the module `MODULE`. The name of the abstract syntax module produced by *rosy* is normally derived from the name of the input file, using an extension of `.py`. The name of the *C* language stub files is derived from the name of the module defined in the input file. Hence

```
rosy module.ry
```

will produce:

`module.py`: the abstract syntax module;

`MODULE-ops.h`: operation definitions and stubs;

`MODULE-ops.c`: data structures for *librosy*(3n); and,

`MODULE-stubs.c`: stub definitions for *lint*(1)

Intuitively, one would suspect that the the rules for *make*(1) should take advantage of the `.ry` and `.py` suffix relationship for a rule like:

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

For reasons explained later on (in Section 5.4.2 on page 53), it is advantageous to use a more complicated set of rules:

```
MODULE-asn.py:    module.ry
                  rosy $(RYFLAGS) -o $@ module.ry
MODULE-ops.c:     module.ry
MODULE-ops.h:     module.ry
MODULE-stubs.c:   module.ry
```

Usually `RYFLAGS` is empty.

In our particular discipline, we will generate two sets of files: one for the invoker and the other for the performer.

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

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

4.4.3 Grinding

Figure 6.7 on page 79 of *Volume One* contains an entry for the *vgrindefs*(5) file, which may be used by the *vgrind*(1) or *tgrind*(1) facility.

Chapter 5

Pepy Optional Structure-Generator

The *posy*(1) program is a compiler for specifications of abstract syntaxes.

For those interested in trivia, *posy* stands for Pepy Optional Structure-generator (YACC-based). The *pepy*(1) program is an ASN.1 compiler, and the *yacc*(1) program, of course, is a compiler-compiler.

5.1 Abstract Syntax Module

The *posy* program reads a description of an *abstract syntax module* and produces the corresponding *C* structures along with an augmented abstract syntax module.

The syntax of the input file corresponds almost exactly to the notation described in [ISO87, CCI88c]. (But see Section 5.3 on page 52 for the details.) As the ASN.1 language is rich in its notation, the rules are not presented herein, though Section 6.2.1 on page 60 of *Volume One* presents an incomplete treatment of the less substantive aspects of the language.

An example of a complete abstract syntax module can be found in Figure 5.1 starting on page 42. This corresponds to the output produced by *rosy*(1) when given the remote operations module in Figure 4.1 as input.

For our purposes, the input file is relatively unimportant as it is usually generated by the *rosy* program. Hence, the output from *posy* is where this chapter focuses.

```
-- automatically generated by rosy 3.1 #15 (killer-rat), do not edit!

IMAGE DEFINITIONS ::=

BEGIN

Initialize-INV ::=
        SEQUENCE {
            proposed[0]
                IMPLICIT Version
                DEFAULT { version-1 }
        }

Initialize-RSP ::=
        SEQUENCE {
            negotiated[0]
                IMPLICIT Version
                OPTIONAL
        }

VersionsSupported ::=
        SEQUENCE {
            supported[0]
                IMPLICIT Version
        }

Version ::=
        BIT STRING {
            version-1(0)
        }

ReadImage-INV ::=
        SEQUENCE {
                SEQUENCE OF
                    CHOICE {
                        file-name[0]
                            IMPLICIT Filename,
```

```
                    mail-address[1]
                        IMPLICIT Mail-Address
                },

            SEQUENCE OF
                Format
        }

ReadImage-RSP ::=
        SEQUENCE {
            CHOICE {
                x-windows
                        X-Windows-Bitmap
            }
        }

FormatsSupported ::=
        SEQUENCE {
            supported[0]
                IMPLICIT Format
        }

Format ::=
        BIT STRING {
            x-windows-bitmap(0)
        }

Filename ::=
    [APPLICATION 1]
        IMPLICIT GraphicString

Mail-Address ::=
    [APPLICATION 2]
        IMPLICIT SEQUENCE {
            local[0]
                IMPLICIT GraphicString,

            domain[1]
                IMPLICIT GraphicString,
```

```
            options[2]
                IMPLICIT BIT STRING {
                    default-local(0),
                    default-host(1)
                }
                DEFAULT { default-local,  default-host }
        }

X-Windows-Bitmap ::=
    [APPLICATION 100]
        IMPLICIT SET {
            width[0]
                IMPLICIT INTEGER,

            height[1]
                IMPLICIT INTEGER,

            hot-spot-x[2]
                IMPLICIT INTEGER
                DEFAULT -1,

            hot-spot-y[3]
                IMPLICIT INTEGER
                DEFAULT -1,

            data[4]
                IMPLICIT OCTET STRING
        }

END
```

Figure 5.1: Example of an Abstract Syntax Module

5.2 POSY Environment

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

5.2.1 C Language Structures

For each type in the abstract syntax module, *posy* will define a corresponding *C* language structure (`struct`).

The rules that *posy* uses to generate the structures are fairly simple:

`BOOLEAN`: represented as a `char`.

`INTEGER`: represented as an `int`. If the `INTEGER` type has a list of tagged values, then for each value:

```
#define int_MODULE_TYPE_TAG value
```

as in:

```
#define int_IMAGE_IntList_zero 0
```

`BIT STRING`: represented as a `struct PElement` (these are described in Section 5.4.5 of *Volume One*). If the `BIT STRING` type has a list of tagged bits, then for each bit:

```
#define bit_MODULE_TYPE_TAG value
```

as in:

```
#define bit_IMAGE_Version_version__1 0
```

In addition, one other definition is present:

```
#define bits_MODULE_TYPE string
```

as in:

```
#define bits_IMAGE_Version "\020\01version-1"
```

`OCTET STRING`: represented as a `struct qbuf` (these are described in Section 5.4.4 of *Volume One* and Section 4.3.2 of *Volume Two*).

`REAL`: represented as a `double`.

`NULL`: represented as a `char`.

`SEQUENCE/SET/ANY`: represented as a `struct PElement` (which are described throughout Chapter 5 of *Volume One*). This format is used for those `SEQUENCE`s or `SET`s which contain no elements.

`SEQUENCE OF/SET OF`: by default, represented as a `struct` with two elements, a linked-list. The first element is the type for set or sequence, the second element is a pointer to another instance of the structure. For example:

```
SEQUENCE OF
    Format
```

might be translated as:

```
struct element_IMAGE_4 {
    struct type_IMAGE_Format *Format;

    struct element_IMAGE_4 *next;
};
```

By using the '`-h2`' option (described below), an array structure can be generated instead.

`SEQUENCE/SET`: represented as a simple `struct` with an element for each member. For example:

```
Mail-Address ::= SEQUENCE {
    local[0]
        IMPLICIT GraphicString,

    domain[1]
        IMPLICIT GraphicString,

    options[2]
        IMPLICIT BITSTRING
}
```

might be translated as:

```
struct type_IMAGE_Mail__Address {
    struct type_UNIV_GraphicString *local;

    struct type_UNIV_GraphicString *domain;

    PE      options;
};
```

`CHOICE`: represented as a `struct` containing an `int` and a `union`.

```
CHOICE {
    file-name[0]
        IMPLICIT Filename,

    mail-address[1]
        IMPLICIT Mail-Address
}
```

might be translated as:

```
struct choice_IMAGE_0 {
  int     offset;
#define choice_IMAGE_0_file__name       1
#define choice_IMAGE_0_mail__address    2

  union {
     struct type_IMAGE_Filename *file__name;

     struct type_IMAGE_Mail__Address *mail__address;
  }   un;
};
```

`OBJECT IDENTIFIER`: represented as a `struct OIDentifier` (these are described in Section 5.4.6 of *Volume One*).

`MODULE.TYPE`: represented as a pointer to a `struct type_MODULE_TYPE`.

Although these rules might seem complicated, in point of fact they are actually very straight-forward and regular. The easiest way to gain familiarity with them is to run some abstract syntax modules through *posy* and then view the resulting definitions. Figure 5.2 starting on page 51 presents the *C* language structures corresponding to the abstract syntax module in Figure 5.1.

Controlling the names that POSY generates

One of the least friendly aspects of *posy* is the algorithm it uses for choosing the names of *C* language structures being generated. The rules for choosing some names are fixed, i.e., the top-level name generated for an ASN.1 type cannot be changed. However, the names used for the parts of the `struct`s generated by *posy* can be controlled — to an extent using either (or both) of two techniques.

The first method takes advantage of the observation that *posy* uses commentary tags in the ASN.1 definition, whenever possible, as variable names. To revisit an earlier example:

```
Mail-Address ::= SEQUENCE {
    local[0]
        IMPLICIT GraphicString,

    domain[1]
        IMPLICIT GraphicString,

    options[2]
        IMPLICIT BITSTRING
}
```

might be translated as:

```
struct type_IMAGE_Mail__Address {
    struct type_UNIV_GraphicString *local;

    struct type_UNIV_GraphicString *domain;

    PE      options;
};
```

In each case, the names of the variables (e.g., `local`) were taken from the commentary tag. Note that if no tag is present, *posy* must choose a name in a different fashion. Hence, the first level of control is to simply add short, descriptive commentary tags to the remote operations module (the *rosy* program will preserve the ASN.1 when it generates an abstract syntax module).

The second method is to augment the remote operations module with the `%[ ... %]` construct. This construct allows the user to select both the name used for intermediate structures that are generated and the name used for the corresponding variable in certain cases. Consider:

```
ReadImage-INV ::= SEQUENCE {
    ...

    SEQUENCE OF %[ format_list $ head %]
        FORMAT

    ...
}
```

might be translated as:

```
struct type_IMAGE_ReadImage__INV {
    ...

    struct format_list {
        struct type_IMAGE_Format *element_IMAGE_4;

        struct format_list *next;
    } *head;

    ...
};
```

More formally, the

```
%[ structure-name $ variable-name %]
```

construct can be used to control the names used when generating intermediate structures. The construct can be used with any of these ASN.1 constructs:

- `SEQUENCE %[ ... %]`
- `SEQUENCE OF %[ ... %]`

- `SEQUENCE %[ ... %] { ... }`
- `SET %[ ... %]`
- `SET OF %[ ... %]`
- `SET %[ ... %] { ... }`
- `CHOICE %[ ... %] { ... }`

If the `$ variable-name` portion of the construct is missing, then only the structure name is taken from the construct, the name of the corresponding variable is generated in the usual fashion.

NOTE: This facility is only allowed in the use of nested constructs. It is not possible to override the names chosen for the top level constructs as this would break the inheritence of structures names from other modules. E.g.,

```
Defn ::= SEQUENCE %[ ... %] { ...
```

is not supported but

```
Defn ::= SEQUENCE {
        SEQUENCE OF %[ ... %] Thing,
        ...
```

is.

5.2.2 Augmented Abstract Syntax Module

In addition to the definition of *C* language structures, *posy* will prepare an augmented abstract syntax module for use with the *pepy*(1) program. With this augmented module, *pepy* will generate *C* code to translate *C* structures to and from their corresponding abstract syntax. As this is entirely an automatic process, the contents of the augmented abstract syntax module are relatively unimportant.

```
/* automatically generated by posy 3.1 #16 (gremlin), do not edit! */

#ifndef _module_IMAGE_defined_
#define _module_IMAGE_defined_

#include <isode/psap.h>
#include <isode/pepy/UNIV-types.h>

#define type_IMAGE_Version      PElement
#define bits_IMAGE_Version      "\020\01version-1"
#define bit_IMAGE_Version_version__1       0

#define type_IMAGE_Format       PElement
#define bits_IMAGE_Format       "\020\01x-windows-bitmap"
#define bit_IMAGE_Format_x__windows__bitmap       0

#define type_IMAGE_Filename     type_UNIV_GraphicString

struct type_IMAGE_Initialize__INV {
   struct type_IMAGE_Version *proposed;
};

struct type_IMAGE_Initialize__RSP {
   struct type_IMAGE_Version *negotiated;
};

struct type_IMAGE_VersionsSupported {
   struct type_IMAGE_Version *supported;
};

struct type_IMAGE_ReadImage__INV {
   struct element_IMAGE_1 {
         struct choice_IMAGE_0 {
            int     offset;
#define choice_IMAGE_0_file__name          1
#define choice_IMAGE_0_mail__address       2

            union {
```

```
                struct type_IMAGE_Filename *file_name;

                struct type_IMAGE_Mail_Address *mail_address;
            }       un;
        } *choice_IMAGE_0;

        struct element_IMAGE_1 *next;
    } *element_IMAGE_1;

    struct element_IMAGE_4 {
        struct type_IMAGE_Format *element_IMAGE_5;

        struct element_IMAGE_4 *next;
    } *element_IMAGE_4;
};

struct type_IMAGE_ReadImage_RSP {
    struct choice_IMAGE_1 {
        int     offset;
#define choice_IMAGE_1_x_windows        1

        union {
            struct type_IMAGE_X_Windows_Bitmap *x_windows;
        }       un;
    } *choice_IMAGE_1;
};

struct type_IMAGE_FormatsSupported {
    struct type_IMAGE_Format *supported;
};

struct type_IMAGE_Mail_Address {
    struct type_UNIV_GraphicString *local;

    struct type_UNIV_GraphicString *domain;

    PE      options;
#define bits_IMAGE_options      "\020\01default-local\02default-host"
#define bit_IMAGE_options_default_local         0
```

```
#define bit_IMAGE_options_default_host   1
};

struct type_IMAGE_X_Windows_Bitmap {
    int     width;

    int     height;

    int     hot_spot_x;

    int     hot_spot_y;

    struct qbuf *data;
};
#endif
```

Figure 5.2: Example of C Language Structures

5.3 Known Deficiences

posy uses essentially the same front-end as the *pepy*(1) program, so it has some limitations in the ASN.1 syntax it can accept. Consult Section 6.3.1 for the details.

When generating the augmented abstract syntax module for *pepy*, *posy* has a well-defined set of rules. There are, however, some aspects which are still incomplete or subject to change:

- On determining if an `OPTIONAL` element is present, for elements represented by a pointer (e.g., an `OCTET STRING` and its corresponding `struct qbuf` pointer), if the pointer is non-`NULL` (non-zero valued), then the element is considered to be present.

- For elements which are represented by a scalar and not a pointer (e.g., an `INTEGER` and its corresponding `int`), an additional element, `optionals`, is created in the structure. This is a bitmask value, with each bit indicating whether a particular element is present or not. So for optional element, a test is first required to see if this element is really present. e.g.,

```
if (element -> optionals & opt_element_member1)
    /* process element -> member1 */
```

5.4 Running POSY

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

5.4.1 Options

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

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

Normally, *posy* ignores all *pepy*-style augmentations except the "verbatim" actions occuring at the very beginning and end of the module. The '`-d`' switch directs *posy* to ignore the verbatim actions as well.

The '`-f`' switch directs *posy* to generate *C* routines to deallocate the structures it defines (e.g., for type `type_MODULE_type`, a routine called `free_MODULE_type` is defined). These are appended to the augmented module definition (as a consequence, use of the '-f' switch forces use of the '-d' switch).

The '-h' switch enables additional heuristics when *posy* generates a *C* language structure definition. These heuristics are used to make the `struct`s more concise. For example, compare the definitions in Figure 5.2 with those in Figure 5.3 starting on page 57. Both were derived from the abstract syntax module in Figure 5.1, but the definitions in Figure 5.3 was generated using the '-h0' switch. The current wisdom is that use of the '-h0' switch is a good thing. Enabling any other option also results in enabling option '-h0'.

The '-h1' switch enables "clever" but non-unique structure naming. Use of the '-h1' switch is **not** recommended.

The '-h2' switch enables the generation of arrays rather than linked-lists whenever possible. For example, compare the definitions in Figure 5.2 with the fragment in Figure 5.4 on page 58. Both were derived from the abstract syntax module in Figure 5.1, but the fragment in Figure 5.4 was generated using the '-h2' switch. The current wisdom makes no pronouncement as to whether the '-h2' switch is a good thing: some people prefer linked-lists, others perfer arrays.

The '-o' switch sets the name of the output file. If this switch is not specified, the standard output is used (*posy* cannot derive the name of the output file from the input file since both should have extension `.py`).

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

5.4.2 Makefiles

By convention, input files to *posy* have the extension `.py`. This causes something of a problem with *make*(1) as the augmented abstract syntax module which *posy* outputs should also have an extension of `py`.

Since *posy* cannot intuit the name of the output file from the input file, rather than starting with file `module.py`, we assume we are starting with file `MODULE-asn.py`. (Hence the reason for the complexity in Section 4.4.2.) Let us arbitrarily select the file named `MODULE-types.py` for the augmented abstract syntax module produced by *posy*. Hence

```
posy -o MODULE-types.py MODULE-asn.py
```

will produce:

`MODULE-types.py`: the augmented abstract syntax module; and,

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

Our *make*(1) rules are:

```
MODULE-types.py:  MODULE-asn.py
                  posy $(POFLAGS) -o $@ MODULE-asn.rpy
MODULE-types.h:   MODULE-types.py
```

Usually `POFLAGS` is set to "`-f -h`".

In our particular discipline, we will generate two sets of files: one for the invoker and the other for the performer.

```
MODULE-Rtypes.c:  MODULE-types.py
          pepy -a PY_advise $(PYFLAGS) -o $@ MODULE-types.py

MODULE-Itypes.c:  MODULE-types.py
                  pepy $(PYFLAGS) -o $@ MODULE-types.py
```

```
/* automatically generated by posy 3.1 #16 (gremlin), do not edit! */

#ifndef _module_IMAGE_defined_
#define _module_IMAGE_defined_

#include <isode/psap.h>
#include <isode/pepy/UNIV-types.h>

#define type_IMAGE_Initialize__INV         type_IMAGE_Version
#define free_IMAGE_Initialize__INV         free_IMAGE_Version

#define type_IMAGE_Initialize__RSP         type_IMAGE_Version
#define free_IMAGE_Initialize__RSP         free_IMAGE_Version

#define type_IMAGE_VersionsSupported       type_IMAGE_Version
#define free_IMAGE_VersionsSupported       free_IMAGE_Version

#define type_IMAGE_Version       PElement
#define bits_IMAGE_Version       "\020\01version-1"
#define bit_IMAGE_Version_version__1       0
#define free_IMAGE_Version       pe_free

#define type_IMAGE_ReadImage__RSP          type_IMAGE_X__Windows__Bitmap
#define free_IMAGE_ReadImage__RSP          free_IMAGE_X__Windows__Bitmap

#define type_IMAGE_FormatsSupported        type_IMAGE_Format
#define free_IMAGE_FormatsSupported        free_IMAGE_Format

#define type_IMAGE_Format        PElement
#define bits_IMAGE_Format        "\020\01x-windows-bitmap"
#define bit_IMAGE_Format_x__windows__bitmap        0
#define free_IMAGE_Format        pe_free

#define type_IMAGE_Filename      type_UNIV_GraphicString
#define free_IMAGE_Filename      free_UNIV_GraphicString

struct type_IMAGE_ReadImage__INV {
    struct element_IMAGE_1 {
```

```
            struct choice_IMAGE_0 {
                int     offset;
#define choice_IMAGE_0_file_name        1
#define choice_IMAGE_0_mail_address     2

                union {
                        struct type_IMAGE_Filename *file_name;

                        struct type_IMAGE_Mail_Address *mail_address;
                }       un;
            } *choice_IMAGE_0;

            struct element_IMAGE_1 *next;
    } *element_IMAGE_1;

    struct element_IMAGE_4 {
            struct type_IMAGE_Format *Format;

            struct element_IMAGE_4 *next;
    } *element_IMAGE_4;
};

struct type_IMAGE_Mail_Address {
    struct type_UNIV_GraphicString *local;

    struct type_UNIV_GraphicString *domain;

    PE      options;
#define bits_IMAGE_options      "\020\01default-local\02default-host"
#define bit_IMAGE_options_default_local         0
#define bit_IMAGE_options_default_host  1
};

struct type_IMAGE_X_Windows_Bitmap {
    int     width;

    int     height;

    int     hot_spot_x;
```

```
    int     hot_spot_y;

    struct qbuf *data;
};
#endif
```

Figure 5.3: Example of C Language Structures (with -h0)

```
...

struct type_IMAGE_ReadImage_INV {
    struct element_IMAGE_1 {
            int nelem;
            struct choice_IMAGE_0 {
                int     offset;
#define choice_IMAGE_0_file_name          1
#define choice_IMAGE_0_mail_address       2

                union {
                        struct type_IMAGE_Filename *file_name;

                        struct type_IMAGE_Mail_Address *mail_address;
                }       un;
            } *element_IMAGE_2[1];
    } *element_IMAGE_0;

    struct element_IMAGE_4 {
            int nelem;
            struct type_IMAGE_Format *Format[1];
    } *element_IMAGE_3;
};
int         free_IMAGE_ReadImage_INV ();

...
```

Figure 5.4: Example of C Language Structures (with -h2)

Chapter 6

Compiling Data-Structures

The *pepy*(1) program is a compiler for the presentation elements discussed in Chapter 5 in *Volume One*. Although experimental in previous releases, the *pepy* program has now stabilized.

For those interested in trivia, *pepy* stands for Presentation Element Parser (YACC-based). The *yacc*(1) program, of course, is a compiler-compiler.[1]

6.1 Warning

Please read the following important message.

NOTE:	Users of *The Cookbook* emply the *posy*(1) program program as the front-end to *pepy*. Hence, this chapter may be safely skipped. It is included only for those misguided individuals who do not *posy*. This behavior is not recommended.

6.2 Syntax Rules

The *pepy* program reads a description of an *abstract syntax module* and produces a *C* program which recognizes the objects described in the module, or generates those objects, or prints those objects. The syntax of the input

[1] This abbreviation is now strictly incorrect, as the *pepy* program can now generate presentation elements as well as parse them.

file corresponds almost exactly to the Abstract Syntax Notation (ASN.1) discussed in Section 6.8 on page 87.

6.2.1 ASN.1 Notation

To summarize the ASN.1 syntax:

- Comments start with two adjacent dashes (‘`--`’) and are terminated by either new-line or another two adjacent dashes, whichever comes first.
- Whitespace between tokens is ignored.
- Keywords are entirely in uppercase.
- The names of rules (types and values) start with an uppercase character and may use either upper- or lowercase characters afterwards.
- The names of context-specific tags start with a lowercase character and may use either upper- or lowercase characters afterwards.
- A name is an alphabetic character followed by zero or more alphanumerics or dashes (except ‘`--`’, of course).
- Literal strings are surrounded by double-quotes (‘`"`’).
- Numbers take on one of two forms:
  - a string of digits, optionally preceded by a minus sign, which is interpreted in decimal; and,
  - a string surrounded by single-quotes (‘`'`’) and immediately followed by one of `B`, `b`, `H`, or `h`, which is interpreted as either a binary (`B` or `b`) or hexadecimal (`H` or `h`) number.

An example of a specification using ASN.1 can be found in Figure 6.1.

6.2.2 ASN.1 Extensions

The *pepy* program recognizes several extensions to the ASN.1 syntax: *compiler directives*, *action statements*, *control statements*, and, *value passing statements*.

```
PERSONNEL DEFINITIONS ::=

BEGIN

PersonnelRecord ::=
        [APPLICATION 0]
          IMPLICIT SET {
                Name,

              title[0]
                IA5String,

                EmployeeNumber,

              dateOfHire[1]
                Date,

              nameOfSpouse[2]
                Name,

              [3]
                IMPLICIT SEQUENCE OF ChildInformation DEFAULT {}
          }

ChildInformation ::=
        SET {
              Name,

          dateofBirth[0]
              Date
        }

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

              initial
                IA5String,

              familyName
                IA5String
          }

EmployeeNumber ::=
        [APPLICATION 2]
          IMPLICIT INTEGER

Date ::=
        [APPLICATION 3]
          IMPLICIT IA5String -- YYYYMMDD --

END
```

Figure 6.1: Example of a specification using ASN.1

Compiler Directives

The compiler directives control the type of *C* routines that are produced. The *pepy* program can generate routines to parse, create, and print presentation elements. The directives may occur after the `BEGIN` keyword; and anywhere an `END` keyword may occur. Each occurrence specifies the type of *C* routine to generate for successive type definitions and applies until the next directive.

The directives are:

`ENCODER` **build** — directs *pepy* to create routines for the construction of presentation elements;

`DECODER` **parse** — directs *pepy* to create routines to parse presentation elements;

`PRINTER` **print** — directs *pepy* to create routines which print the contents of presentation elements; or,

`SECTIONS` **build parse print** — directs *pepy* to create multiple forms of above.

The `build`, `parse`, and `print` strings are used to construct the name of the routine and are described later.

With the `SECTIONS` directive, the value `none` may be used as a null-valued place-holder. For example:

```
SECTIONS none undo pp
```

indicates that *pepy* should generate decoding and printing routines (using prefixes `undo` and `pp` respectively), and that no encoding routines should be generated.

There is also another directive, which must occur immediately *before* the `BEGIN` keyword:

`PREFIXES` **build parse print** — tells *pepy* what `SECTIONS` were used in other modules.

Action Statements

The lexemes `%{` and `}%` surround *C* code which the resulting program should execute at the appropriate time. The code may be of arbitrary length, spanning many lines in the input file.

Actions may occur in several places in the input file. Depending on the position of the actions, slightly different semantics are associated with the code. There are four types of actions: *verbatim actions*, *type actions*, *tag actions*, and, *declaration actions*.

The first type of actions are *verbatim* actions which occur in two places in the input file:

- immediately before the `BEGIN` keyword; and,
- after the `END` keyword in the input file.

These actions are copied strictly verbatim, being intended for definitions, subroutines, and the like.

The second type of actions are *type* actions which are placed after a type or sub-type definition, and after each element in a "named number". In the former case, please note that the action must be placed before the `OPTIONAL` and `DEFAULT` lexemes. During the copy, the lexeme `$$` is expanded to represent the object currently being examined. The exact expansion depends upon the element being processed and what sort of processing is being done. In decoding and printing routines, `$$` will expand to a `PE` in actions before the element is recognized and to the actual data after the processing. In the case of encoding routines, the opposite is true. The precise nature of the data is shown below.

Type	**C declaration**
`BOOLEAN`	`int`
`INTEGER`	`int`
`BITSTRING`	`PE` (a bitvector)
`OCTETSTRING`	`char *` (length is `$$_len`)
`REAL`	`double`
any `SEQUENCE`	`PE` (a sequence)
any `SET`	`PE` (a set)
`OBJECT␣IDENTIFIER`	`OID`

Consider the type definition in Figure 6.2:

- `action1` will be executed only if the object being recognized is a `document`;
- `action2` will be executed only if the object being recognized is a `paragraph`; and,
- `action3` will always be executed afterwards.

The third type of actions are *tag* actions which are placed after the tag for a tagged type or after the name for a named type. Consider the type definition in Figure 6.3:

```
LogicalObjectType ::=
        INTEGER {
                document(0)
                %{ action1 %},

                paragraph(1)
                %{ action2 %},
        }
        %{ action3 %}
```

Figure 6.2: Example of a TYPE action

```
ProtocolElement ::=
        CHOICE {
           textUnit[8]
                IMPLICIT TextUnit
                %{ action1 %},

           specificLogicalDescriptor[5]
                %{ action2 %}
                IMPLICIT LogicalDescriptor
        }
```

Figure 6.3: Example of a TAG action

- `action1` is executed after `textUnit` has been fully processed as a `TextUnit`; but,

- `action2` is executed after `specificLogicalDescriptor` has been recognized, but before it has been processed as a `LogicalDescriptor`. This type has both a name (`specificLogicalDescriptor`) and a tag (`[5]`). Although two actions are present, one only action is honored, and this action should be specified after the tag.

A tag action may also be specified for `SET␣{`, `SEQUENCE␣{` and `CHOICE␣{` – even when they are not tagged. The action is placed before the left brace, and is executed when the `SET`, `SEQUENCE`, or `CHOICE` is recognized but before any of its members are processed.

A tag action may additionally be specified directly after the "`::=`" lexeme, even when not tagged. This has the same effect as the tagged rule. This is the only way to provide a pre-action for untagged single rules.

Finally, a tag action may occur after the phrase `SET␣OF` or `SEQUENCE␣OF`. This action is executed before processing each instance of the base type of the `SEQUENCE` or `SET`. This is useful for allocating memory for each of the member elements.

The final type of actions are declaration actions. These are designed to allow the declaration of variables specific to a generated function. Such actions occur between the type name and the "`::=`" lexeme. The statements placed here are appended to the declaration section of the generated routine. Only variable definitions should be placed here, as in:

```
X400 %{ char cp[100]; %} ::=
    SET {
        X4001984 %{ strcpy(cp, "1984"); %},
        X4001988 %{ strcat(cp, " 1988"); %} OPTIONAL,
        X4001992 %{ strcat(cp, " 1992"); %} OPTIONAL
    }
    %{ puts(cp); %}
```

Control Statements

When constructing routines for encoding presentation elements, additional constructs are required at certain points to control the choices available. The control constructs are specified between the lexemes `<<` and `>>`. The control itself consists of an appropriate *C* statement which takes a different form for each of the constructs. The affected types are listed below.

- `CHOICE` — to decide which of the selection to choose. The statement consists of an expression of integer value that chooses the n'th value. For example in

  ```
  NewDef ::=
      CHOICE << 2 >> {
          [0] FirstElement,
          [1] SecondElement,
          [2] ThirdElement
  ```

```
    }
```

the control `<< 2 >>` would choose the second type in a choice, which is not necessarily the element tagged with `[2]`, as in this case. The statement always precedes the opening brace of the `CHOICE`.

- `SEQUENCE␣OF` and `SET␣OF` — to indicate the number of elements require. This construct has the syntax of a for loop in C. Elements are added to the `SEQUENCE/SET` while the second expression yields `TRUE`. The sequence

  ```
  SeqDef ::=
      SET OF << count = 0; count < 10; count ++ >>
          MatchingAccessories
  ```

  would produce ten elements in the set. The statement follows the `OF` and any optional actions.

- `OPTIONAL` — to decide whether the optional element should be included. This statement follows the `OPTIONAL` keyword. It is a simple boolean expression. If the value returned is `TRUE` then the element is included.

- `DEFAULT` — to decide whether the optional element should be included. This statement follows the `DEFAULT` keyword/value pair. It is a simple boolean expression. If the value returned is `TRUE` then the element is included.

  Arguably, this should be unnecessary — *pepy* should look at the value immediately following `DEFAULT` to decide if the element need be encoded.

The example in Figure 6.4 should make this a little clearer.

Value Passing Statements

A number of extensions are available to assist in passing values between the generated routines. The generated routines pass these values between rules as parameters to each routine. This can obviate the need for much of the *C* code associated with each type.

```
ENCODER build

Example ::=
        SEQUENCE {
            -- select maybe element if present
            Maybe
            OPTIONAL <<maybe == PRESENT>>,

            SEQUENCE OF
                -- build elements
                %{ construct_elements(); %}
                -- now produce them
                <<init_elements(); more_elements();>>
                BizarreEvents,

            CHOICE
            %{ choose = leap_year() ? 2 : 1; %}
            <<choose>>
            {
                [0] MyProposal,
                [1] YourProposal
            }

            INTEGER
            [[i defvalue]]
            DEFAULT 4 <<defvalue != 4>>
        }
```

Figure 6.4: Examples of CONTROL actions

To allow values to be passed to and obtained from these parameters, value tokens can be associated with the type definitions. These tokens are incorporated in the lexemes `[[` and `]]`. The tokens come in two types. The first is a key letter followed by a expression. The second is the same with an additional parameter to specify the length of the first parameter. These formats are shown below.

- `[[b expression]]` — a boolean. For encoding, this is the value that the presentation element will take. For decoding and printing, the value of the presentation element is assigned to this variable.
- `[[i expression]]` — an integer. For encoding, this is the value that the presentation element will take. For decoding and printing, the value of the presentation element is assigned to this variable.
- `[[s expression]]` — a string. For encoding, this is the value that the presentation element will take, a null-terminated string. For decoding and printing, the string value of the presentation element is assigned to this variable; the user is responsible for freeing the storage assigned with `free`.
- `[[O expression]]` — an object identifier. For encoding, this is the value that the presentation element will take, an `OID` value (see Section 5.4.6 on page 135 in *Volume One*). For decoding and printing, the `OID` value of the presentation element is assigned to this variable; the user is responsible for freeing the storage assigned with `oid_free`.
- `[[p expression]]` — a user supplied parameter, `parm`. This parameter is of type `PEPYPARM`, which can be `#define`'d by the user to a suitable value. If not defined, it defaults to a `char␣*`. Most commonly this type is defined to be a union of all parameter types that need to be passed (but see the next item).
- `[[P type]]` — a *C* type which defines the type of the parameter for this definition. If present, this construct must occur directly after the definition name.
- `[[o expression1 $ expression2]]` — a byte pointer/length pair. For encoding, these indicate the byte value and length of the presentation element, which needn't be null-terminated. For decoding and

printing, the byte value and length are assigned to these variables; the user is responsible for freeing the storage assigned with `free`.

- `[[q expression]]` — a `qbuf`. For encoding, this is a pointer to a `qbuf` (described in Section 5.4.4 in *Volume One*) which is used to build the presentation element. If the `qbuf` contains only one element in the linked list, then a primitive element is built, otherwise a constructed type is built. which needn't be null-terminated. For decoding and printing, this value is undefined.

- `[[x expression1 $ expression2]]` — a bit pointer/length pair. For encoding, the value of the presentation element is derived by supplying `expression1` and `expression2` as the first two arguments to `strb2bitstr` (described in Section 5.4.5 in *Volume One*). For decoding and printing, the bitstring value and length are assigned to these variables; the user is responsible for freeing the storage assigned with `free`.

- `[[a expression]]` — a presentation element. For encoding, this is the value that the presentation element will take. For decoding and printing, the presentation element is assigned to this variable.

- `[[r expression]]` — a real number. For encoding this is a floating point number that will be encoded. For decoding this represents a floating point number variable in double format.

The example in Figure 6.5 should demonstrate these features.

```
SalaryDefs DEFINITIONS ::=

%{
#include <stdio.h>

char   *myname;

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

#define PEPYPARM struct salary_record *


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

    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, &salary);

    exit (0);
}
%}

...
```

Figure 6.5: Examples of Parameters in Pepy

```
...

BEGIN

SECTIONS build unbuild none

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

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

TheSalary ::=
        INTEGER

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.

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`.

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 85 for an example of the `advise` procedure.

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

```
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 85 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.

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

#include <stdio.h>

int     print_PERSONNEL_PersonnelRecord ();


main (argc, argv, envp)
int     argc;
char  **argv,
      **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:

The '`-a`' switch sets the name of the routine used by the `advise` facility (see page 74).

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) $<
```

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 61. (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.

```
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

```
-- personnel record pretty-printer

PERSONNEL DEFINITIONS ::=

%{
#define DEBUG

#include <stdio.h>


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


static char *myname = "prpp";


void    adios ();


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"
};

...
```

Figure 6.8: Pretty-Printer initialization

```
...

/*    MAIN */

/* ARGSUSED */

main (argc, argv, envp)
int         argc;
char  **argv,
      **envp;
{
    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");

    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);

        pe_free (pe);
    }

    ps_free (ps);

    exit (0);
}

%}

...
```

Figure 6.8: Pretty-Printer initialization (continued)

```
...

BEGIN

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

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

                EmployeeNumber,

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

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

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

Figure 6.9: Pretty-Printer top-level

```
...

ChildInformation ::=
        SET {
                Name
                %{
                   printf ("\nChild Information\nName:  %s %s.  %s\n",
                         givenName, initial, familyName);
                %},

           dateofBirth[0]
                Date
                %{ printf ("Date of Birth:  %s\n", Date); %}
        }

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

                initial
                   IA5String [[s initial]],

                familyName
                   IA5String [[s familyName]]
           }

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

...
```

Figure 6.10: Pretty-Printer type definitions

```
...

Date ::=
        [APPLICATION 3]
          IMPLICIT IA5String -- YYYYMMDD --
          [[s Date]]
          %{
                int year, month, day;
                static char buffer[BUFSIZ];

                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

...
```

Figure 6.10: Pretty-Printer type definitions (continued)

```
...

%{
/*    DEBUG */

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

    switch (debug) {
            case NOTOK:
                break;

            case OK:
                if ((debug = (cp = getenv ("PEPYDEBUG")) && *cp ? atoi (cp)
                                    : 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)
                        (void) pe2pl (ps, pe);
                fprintf (stderr, "--------\n");
                ps_free (ps);
                break;
    }
}
#endif

...
```

Figure 6.11: Pretty-Printer finalization

```
...

/*    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)

6.8 For Further Reading

The Abstract Syntax Notation One is specified in [ISO87]. The corresponding CCITT recommendation is defined in [CCI88c].

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. These tables replace the work done by the *pepy* program.

Note that unlike *posy*, *pepsy* does not accept arbitary annotations in the abstract syntax module. It should be raw ASN.1 or the strictly limited augmentations of ASN.1 described later, 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) $<
```

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 43).

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 52 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 *pepsy* to augment the `#include` file it generates with commentary text.

The '-d' switch directs *pepsy*, to always ignore all *pepy*-style augmentations, including the "verbatim" actions occuring at the very beginning and end of the module.

The '-f' switch directs *pepsy* 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 52. Note the '-h2' does not work for pepsy (yet).

Normally, *pepsy* 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
```

7.5 Pepy Enhancements to Pepsy

pepsy supports a limited subset of the *pepy* augmented ASN.1 grammar. This is for efficiency and to reduce the amount of work required to change from *pepy* to *pepsy*. See the file *pepsy/t3.py* for examples of these enhancements.

pepsy differs from pepy in one very important mannar. There is only one version of the ASN.1 grammar which contains all the augmentations for encoding and decoding. In contrast pepy has one definition for each task and consequently the two functions can function very differently. One common problem when turning *pepy* code into *pepsy* code is that the decoding and encoding functions used in the *pepy* version have calling conventions different to those required by *pepsy*. *pepsy* requires that the encoding function be passed a pointer to the C data type carrying the data and the decoding function be passed the *address* of the pointer. There are no such restrictions in *pepy* and often people use other methods, these need to be thoroughly checked and fixed up otherwise obscure core dumps are sure to occur.

pepsy also provides arbitary actions which can reference the encoded and unencoded data. Also provides the ability to slot in functions for handling particular types. Of *pepy*'s augmented ASN.1 the most efficient and encouraged are the value passing statements. This are compiled into very compact table entries and yield the best benefit of pepsy, its small size and corresponding compile times.

7.5.1 Verbatim Actions

Verbatim actions are supported exactly as per *pepy*. The verbatim actions are placed in the `MODULE\_tables.c`, at the start or the end as appropriate. This allows files to be conviently included that contain definitions for types used later in the augmented grammar. It is also convient place for functions that are only relevant to the encoding or decoding of that modules data structures.

7.5.2 Value Passing Statements

These are a restriction of the *pepy* grammar for value passing statements. The C expressions are limited to specifying the field of the C structure carrying data corresponding to that typereference. See Figure 7.1 for example.

Here the C data type carrying the data from which the data is encoded from or decoded into is specified by the

```
[[ P struct pepy_refs * ]]
```

```
Pepy-refs [[ P struct pepy_refs * ]] ::= SEQUENCE {
   [1] IMPLICIT INTEGER [[ i parm->t_int ]],
   [2] IMPLICIT ENUMERATED [[ i parm->t_enum ]] {
      orange(1), apple(2), pear(3), bannana(4) },

   [3] IMPLICIT OCTET STRING [[ q parm->t_qbuf ]],
   [4] IMPLICIT OCTET STRING [[ s parm->t_string ]],
   [5] IMPLICIT OCTET STRING [[ o parm->t_ostring $ parm->t_olen ]],
   [6] IMPLICIT BIT STRING [[ t parm->t_pe ]],
   [7] IMPLICIT BIT STRING [[ x parm->t_bstring $ t_blen ]],
   [8] IMPLICIT BOOLEAN [[ b parm->t_bool ]],
   [9] IMPLICIT OBJECT IDENTIFIER [[ O parm->t_oid ]],
   [10] IMPLICIT REAL [[ r parm->t_real ]],
   [11] ANY [[ a parm->t_any ]],
   [12] Def-types [[ p parm->t_def ]],
   [13] Opt-types [[ p parm->t_opt ]] OPTIONAL

   }
```

Figure 7.1: Pepy value passing

augmentation. This is very useful as it allows the programmer to design their own data structures instead of being forced to use those produced by *pepsy*. This is especially useful for people changing over from another encoder/decoding system which uses different data structures to that of *posy* or *pepsy*. It is still restrictive in that integers must be passed in ints and booleans must be passed in chars but is considerable more flexible than *posy*.

All the value passing types are supported with some being interpreted slightly differently and some news ones. The usage is the same as pepy except, as mentioned before, the C expressions are limited to structure fields.

The name of the pointer to the structure is not used at all and in fact the C expression can be abreviated to just the name of the field which contains the data. Note if the name of a C type is not specified for the typereference pepsy assumes the C type it generates for the typereference. It always generates the definitions for these typereferences in the *-types.h file regardless of whether another type is specified by a `P` value passing statement in the grammar.

- `[[ t field]]` — a bit vector. This is used for BIT STRING types and takes a bit vector as described in Section 5.4.5 in *Volume One*. Note that a bitvector is *not* the same as a Presentation Element and failing to convert between them properly will give inconsistant results.

- `[[p field]]` — a pointer type. value passing statement is interpreted slightly differently to its pepy usage. This specifies the field which should be treated as a pointer to another C type which carries the information corrresponding to that ASN.1 typereference. The structure containing the data can be the current structure, that is to say the data is contained in fields of the current structure and so it shouldn't follow any pointer to it. This is specified by putting a `*` in the field arguement.

- `[[T type $ field]]` — a compact way of specifying a `P` and a `p` value passing statements. This is essential for contructed types like sequence and set which need both value passing statements but the normal (long) form is not suitable. If the Information in a contructed form is carried in a different type, perhaps a substructure, there is no way to specify this new type and how to get to it from the existing data structure.

- `[[E fn-name]]` — Encoding function

- `[[D fn-name]]` — Decoding function These entries give the name of a function which encodes/decodes that particular type. This allows people to get right in if they want and hand code routines for encoding/decoding particular types. Perhaps for efficiency or convience when converting from *pepy*. As *pepy* generates functions these could be suitable modified to conform with the convention outlined below and just included in a verbatim action. This may allow a particularly tricky piece of pepy code to be conviently carried over. Obviously if this is used every where the *pepsy* benefit of order of magnitude reduction in code size will be lost, so it is worth avoiding if possible.

Note that the type *must* match that of the ASN.1 being encoded/decoded. This means that the ASN.1 real will expect a double as it field and a boolean expects a char. Anything different will behave in a machine dependant mannar!

7.5.3 Constructed Types - Sequence and Set

As described above these ASN.1 types need to have a `T` value passing statement when used inside a type reference if they are not using the *pepsy* generated C data structure. See Figure 7.2 for example. The outer most sequence or set does not need them of course. Notice in the example that all the `T` fields are using the parent data structure for their information. This illustrates that `*` and `*parm` are considered equivalent, later case is supported for compatility with the augmented ASN.1 grammar produced by *posy*.

Figure 7.3 shows an example with a different C type used for the SEQUENCE OF and SET OF to that specified in the `P` value statement. In the case of the repeating types it is necessary to have a link list to carry the information, with a *next* pointer a part of the structure. The *next* field is indicated by placing it inside the `<<` and `>>` backets as shown. If this control statement is left out it will assume there is a field called "next" in the current structure which does the job. This is consistent with what *posy* generates. Note in the component label `b` the repeated type is a defined type and its data must be carried in the `rep_elem` structure as the encoding/decoding routine is called with the current (`rep_elem`) structure.

```
   Single-type [[ P struct pepy_refs1 * ]] ::= SEQUENCE {
      a SEQUENCE [[ T struct pepy_refs1 * $ * ]] {
            [1] IMPLICIT INTEGER [[ i parm->t_int ]] OPTIONAL << opt_set $ O
PT_INT1 >>
         },
      b SEQUENCE [[ T struct pepy_refs1 * $ * ]] {
            [2] INTEGER [[ i parm->t_int1 ]] OPTIONAL << opt_set $ OPT_INT2
>>
         },

      c SET [[ T struct pepy_refs1 * $ *parm ]] {
         [3] IMPLICIT ENUMERATED [[ i parm->t_enum ]] {
orange(1), apple(2), pear(3), bannana(4) } OPTIONAL << opt_set $ OPT_ENUM1 >>
      },
      d SET [[ T struct pepy_refs1 *  $ *parm ]] {
         [4] ENUMERATED [[ i parm->t_enum1 ]] {
orange(1), apple(2), pear(3), bannana(4) } OPTIONAL << opt_set $ OPT_ENUM2 >>
      }
   }
```

Figure 7.2: Constructed Type Example

```
Repeated [[ P struct repeats * ]]::= SEQUENCE {
   a SEQUENCE OF [[ T struct rep_int * $ rp_sq1 ]]
      <<r>> [1] INTEGER [[ i i ]],

   b SEQUENCE OF [[ T struct rep_elem * $ rp_sq2 ]] <<r_next>>
         Rep-elem [[ p * ]],

   c SET OF [[ T struct rep_int * $ rp_st1 ]] <<r>> INTEGER [[ i i ]],

   d SET OF [[ T struct rep_elem * $ rp_st2 ]] <<r_next>>
         [3] IMPLICIT Rep-elem [[ p * ]],

   e CHOICE [[ T struct repeats  * $ * ]] <<rp_choice>> {
         [1] INTEGER [[ i rp_int ]],
         [2] BOOLEAN [[ b rp_bool ]],
         [3] OCTET STRING [[ o rp_ostring $ rp_int ]]
      }
   }
```

Figure 7.3: Repeated Type Example

7.5.4 Choice

A Choice is another example of a constructed type and so requires `T` value statement. A Choice also needs an integer field for the selection value to go. In encoding this is used to determine which type to choose and for decoding it is set to indicate which type was found. It is used just like pepy with value of 1 meaning the first type and 2 the second and so on. The selection field must be a field of the current structure and if it is left out will default to the *posy* generated value of "offset", which will generate syntax errors if this field is not defined when the corresponding tables field is compiled. There is an example of a choice in the previous Figure 7.3.

There is another "pepy-like" method of specifying which choice to take, which is illustrated in figure 7.4. The syntax allows a C expression to specify which choice to take when encoding and for decoding a C lvalue of an integer in which to store the choice present in the data. If the choice taken in decoding is not important or can be worked out from other information zero can be specified for the lvalue and the choice taken will not be recorded which is a bit more efficient. This is to support people changing *pepy* code and allows a way to those difficult bits which just can't be done by the first method. It is not as efficient as the first as it obviously requires some C code to be especially evaluated for the choice.

7.5.5 Default

Most defaults are handled transparently in *pepsy*. The user fills in what ever value required and *pepsy* will detect if that value is the default or not and encode it appropriately. Similarly on decoding it will decode into a normal value. The cases where it doesn't do this are constructed types. This has only been a problem so far with the default value being the empty sequence or set. At the moment the work around for this problem is to change the default to optional and have the data carried in another structure. To the default null value is carried by setting the pointer the structure to null. This is pretty much the way *posy* handles these situations. Note: you can not specify a default value for an Object Identifier at the moment, again use a work around based on OPTIONAL.

```
SearchResultData [[P struct ds_search_result *]]
    ::=
    CHOICE <D<(*parm)->srr_correlated>>
    <E<((parm->srr_correlated == FALSE) ? 2 : 1)>>
    {
    searchInfo
        SET [[ T struct ds_search_unit * $ srr_un.srr_unit ]]
......
        },
    uncorrelatedSearchInfo
        [0] SET OF [[ T struct ds_search_result * $ srr_un.srr_parts ]]
          <<srr_next>> SearchResult [[p *]]
    }
```

Figure 7.4: Pepy-like selector in a choice

7.5.6 Optional

Optionals are handled in several ways. The original mannar used in *posy* which is a null pointer for those types specified by a pointer and a bit in an integer field called `optionals`. The first optional being specified by the bit 01, the 2nd by 02, the third by 04 and so on. The *pepsy* generated data structures still contain this field and #defines for the various bits if the ASN.1 type requires it, such as INTEGER or ENUMERATED type.

The next method for handling optionals is a generalisation of the *posy* scheme. The user may specify a bit out of an array of bytes (chars) which specifies whether or not the element is present. To use this technique you add after the OPTIONAL and before the `,` or terminating `}` the follow:

```
<< char_array $ Bit_Number>>
```

Where `char_array` is the character array field and `Bit_Number` is actual number of the bit used for this type, usually a #define symbol. Figure 7.2 shows some use of this form of OPTIONAL.

The third and final method of supporting optional is modeled on that used by pepy. It is the same as that used by the "pepy-like" choice selector which allows a C expression for encoding and an integer lvalue for decoding.

```
....
[30] ServiceControls [[p &parm−>cma_common.ca_servicecontrol]]
                 −− DEFAULT {} ,
                OPTIONAL <D<0>>
                <E< (
                (parm−>cma_common.ca_servicecontrol.svc_options != 0) ||
                (parm−>cma_common.ca_servicecontrol.svc_prio != SVC_PRIO_MED
) ||
                (parm−>cma_common.ca_servicecontrol.svc_timelimit != SVC_NOT
IMELIMIT) ||
                (parm−>cma_common.ca_servicecontrol.svc_scopeofreferral != S
VC_REFSCOPE_NONE)
                ) >>,

....
```

Figure 7.5: Optional C-expression/lvalue

As in the choice case it is not as efficient as the previous two methods but is much more general. But for optionals the meaning of the integer is slightly different, non zero means present and zero means absent. Figure 7.5 shows an example of this optional. In this case the the user is not interested in being notified if it is present in the decoded and so specifies zero for the lvalue.

7.5.7 Encoding and Decoding Functions

Figure 7.6 shows the usage of these functions. By placing the `[[X fn ]]` after the type reference the handling for encoding or decoding can be passed to a user written function, called **fn**. The encoding is specified if **X** is an **E** and decoding is specified by an **D**. If only one of encoding or decoding is specified then the task left unspecified is done normally, see in the example the type E-ANY, which is encoded by calling the user function `get_pe()` and decoded by the value passing statement `[[ a * ]]`.

The definition of the encoding and decoding functions must follow a very precise calling and return value conventions. These are not the same as

```
E−ANY ::= ANY [[ a * ]] [[ E get_pe ]]

D−ANY ::= ANY [[ a * ]]  [[ D put_pe ]] [[ F fr_pe ]]

DE−ANY ::= ANY [[ a * ]]  [[ D put_pe ]] [[ E get_pe ]]

 CodeOpt [[ P struct codedata * ]] ::= SEQUENCE {
     a [PRIVATE 5] BasicANY [[ p cd_a ]] OPTIONAL,
     b [35] E−ANY [[ p cd_b ]] OPTIONAL,
     c [45] EXPLICIT D−ANY [[ p cd_c ]] OPTIONAL <<cd_opt_set $ CD_C>>,
     d [1] DE−ANY [[ p cd_d ]] OPTIONAL <<cd_opt_set $ CD_D>>,
     e [15] IMPLICIT INTEGER [[ i cd_int ]] [[ E get_int ]] [[ D put_int ]]
             OPTIONAL << cd_opt_set $ CD_INT >>,
     f GraphicString [[ s cd_string ]] [[ E get_string ]][[ D put_string ]]
             OPTIONAL
 }
```

Figure 7.6: Example of Encoding/Decoding functions

```
get_int(parm, ppe)
struct codedata *parm;
PE      *ppe;
{
   if (ppe == (PE *)0) {
      if (parm->cd_int == 1)  /* Default case */
         return (OK + 1);
      return (OK);
   }

   *ppe = int2prim(parm->cd_int);
   return (OK);
}
```

Figure 7.7: Definition of an Encoding function

those used by *pepy* generated functions which requires lots of parameters, but it shouldn't be too hard to convert *pepy* generated functions into this form. Figure 7.7 shows the definition of a simple but complete example of encoding function.

The encoding function is passed a pointer, **parm**, to the structure that contains the data at the level of the encoding function and the address of a PElement pointer, **ppe**. It expects the user to encode the data present some where in the data structure into a PElement and set the pointer to this PElement. If everything works it should return OK, if something fails it should return NOTOK, to abort the whole encoding.

The OPTIONAL and DEFAULT need to be detected by the function and it should return `OK + 1` and not encode anything. In the example it checks for the value 1 which is the default value and returns correctly otherwise encodes the value and returns `OK`. Note: the tag and class do not need to be set as pepsy will take care of getting that correct. This allows the same function to be used in several places, with different tags.

The decoding function also takes two pointers as arguements and returns `OK` on success and `NOTOK` on failure. The two pointers are different versions of the encoding function pointers, this time the first pointer is the address of the pointer to the decoding function and a pointer to the PElement containing

```
put_int(parm, pe)
struct codedata **parm;
PE      pe;
{
    if (pe)
        (*parm)->cd_int = prim2num(pe);
    else
        (*parm)->cd_int = 1;    /* Default */
    return (OK);
}
```

Figure 7.8: Definition of a Decoding function

the encoded data. The function is expected to construct the decoded data type which may include allocating space for it and setting pointers in the `*parm` data structure to the allocated space. If the PElement pointer is null (there is no encoded data) it means the type is absent and it is up to the function to handle it. This would be the case if it were DEFAULT or OPTIONAL element.

7.5.8 Pepsy C Action Statements

7.5.9 External Type References

Part IV

Recipes

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.

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);

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.

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;

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;
```

```
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:

`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;

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:

`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,
```

```
        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;

`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.

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,
```

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

The parameters to this procedure are:

sd: the association-descriptor;

id: the ID of the invocation being responded to;

err: the error code being returned;

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

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

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

The routine RyDsUReject is used to reject an invocation.

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

The parameters to this procedure are:

sd: the association-descriptor;

id: the ID of the invocation being rejected;

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

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

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

8.7 Waiting for Events

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

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

The parameters to this procedure are:

`sd`: the association-descriptor;

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

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

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

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

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

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

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

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

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

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

8.8 Miscellaneous Routines

There are a few miscellaneous routines of interest.

8.8.1 Association Termination

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

```
int     RyLose (sd, roi)
int     sd;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor; and,

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

8.8.2 Utility Routines

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

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

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

The parameters to this procedure are:

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

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

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

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

The parameters to this procedure are:

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

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

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

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

The parameters to this procedure are:

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

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

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

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

The parameters to this procedure are:

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

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

8.9 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `roi` parameter given to the routine. The element called `roi_preject` in the `RoSAPindication` structure encodes the reason for the failure. To determine the reason, one coerces a pointer to a `RoSAPpreject` structure, and consults the `rop_reason` element of this latter structure. This element can be given as a parameter to the routine `RoErrString` which returns a null-terminated diagnostic string.

```
char   *RoErrString (c)
int     c;
```

8.10 Compiling and Loading

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

Chapter 9

Boilerplate for Initiators

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

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

9.1 Embedded Initiator

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

9.1.1 Association Establishment

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

The routine `AcAssocRequest` is called to establish an association for the service. The arguments are strictly boilerplate: they will work unaltered for most applications. If the association is established, then the routine `RoSetService` is used to tell the remote operations library to use the presentation service as its underlying service.

```
...

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

...

    int     sd;
    struct SSAPref sfs;
    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;
    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);
    if ((pa = aei2addr (aei)) == NULLPA)
        adios (NULLCP, "address translation failed");

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

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

    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));

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

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

...
```

9.1.2 Operation Invocation

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

Synchronous Invocation

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

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

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

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

```
...

    int     result,
            response;
    struct type_MODULE_argument *arg;
    caddr_t out;
    struct RoSAPindication rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

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

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

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

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

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

                        free_MODULE_result (res);
                        break;
                    }

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

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

                        free_MODULE_parameter (parm);
                        break;
                    }

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

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

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

```
            /* NOTREACHED */

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

    free_MODULE_argument (arg);

...
```

ASynchronous Invocation

```
...

int      do_result (), do_error ();

...

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

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

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

        case OK:
            break;

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

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

    free_MODULE_argument (arg);

...
```

```
do_result (sd, id, dummy, result, roi)
int      sd;
int      id,
         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;
}

...


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

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

            break;
         }

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


         default:
```

```
            adios (NULLCP, "unknown error returned (%d)", error);
            /* NOTREACHED */
    }

    /* "parameter" will be automatically free'd when do_error returns */

    return OK;
}

...
```

9.1.3 Association Release

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

```
...

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

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

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

    ACRFREE (acr);

...
```

9.1.4 Error Handling

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

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

`ros_advise`: used to report a ROS error;

`acs_adios`: used to report an ACS error and terminate;

`acs_advise`: used to report an ACS error;

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

`advise`: used to report an error.

It is assumed that there is a definition of the form

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

for use by the `advise` routine.

```
...

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

   _exit (1);
}


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

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

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

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


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

    va_start (ap);

    _advise (ap);

    va_end (ap);

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

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


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

    va_start (ap);

    _advise (ap);

    va_end (ap);
}
```

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

    asprintf (buffer, ap);

    (void) fflush (stdout);

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

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

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

9.2 Interactive Initiator

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

9.2.1 Include File

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

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

#include <isode/rosy.h>


static struct dispatch {
    char   *ds_name;
    int     ds_operation;

    IFP     ds_argument;
    IFP     ds_free;

    IFP     ds_result;
    IFP     ds_error;

    char   *ds_help;
};


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

int     ryinitiator ();
```

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.

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,
```

```
          sd;
char     buffer[BUFSIZ],
         *vec[NVEC + 1];
register struct dispatch   *ds;
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,
         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)
    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");

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

```
if ((pci = ode2oid (mypci)) == NULLOID)
        adios (NULLCP, "%s:  unknown object descriptor", mypci);
if ((pci = oid_cpy (pci)) == NULLOID)
        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);
}

if (argc < 3) {
        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]);
        (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]);

        iloop = 0;
}
```

```
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) {
    if (iloop)
        printf ("failed\n");

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

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

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

if (iloop) {
    for (;;) {
        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) {
            advise (NULLCP, "unknown operation \"%s\"", vec[0]);
            continue;
        }
```

```
            invoke (sd, ops, ds, vec + 1);
        }
    }
    else
        invoke (sd, ops, ds, argv + 3);

    (*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;
{
    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;

    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 */
            break;
```

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

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

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

/*  */

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

    if (sticky) {
	sticky = 0;
	return NOTOK;
    }

    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;
	    }
```

```
            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);
}
```

```
/*  */

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;
```

```
    va_start (ap);

    _advise (ap);

    va_end (ap);

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

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


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

    va_start (ap);

    _advise (ap);

    va_end (ap);
}


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

```
    asprintf (buffer, ap);

    (void) fflush (stdout);

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

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

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

9.2.3 An Example

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

```
...

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

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

   NULL
};


/* ARGSUSED */

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


/* ARGSUSED */

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

```
int     sd;
struct dispatch *ds;
char  **args;
caddr_t *dummy;
{
    struct AcSAPrelease acrs;
    register struct AcSAPrelease   *acr = &acrs;
    struct AcSAPindication  acis;
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort *aca = &aci -> aci_abort;

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

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

    ACRFREE (acr);

    exit (0);
}
```

Chapter 10

Boilerplate for Responders

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

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

10.1 Static Responder

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

Before proceeding however, let's consider what an `#include` file, say `ryresponder.h`, might look like. First, the standard *librosy*(3n) definitions are included, along with the definitions for the daemon logging package. Next, a `dispatch` structure is defined along with the boilerplate routines. The `dispatch` structure will be used by the boilerplate to invoke a user-supplied routine that will respond to an invocation.

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

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


static struct dispatch {
    char    *ds_name;
    int     ds_operation;

    IFP     ds_vector;
};


extern int  debug;


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

int     ryresponder ();
```

10.1.1 Association Management

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

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

The parameters to this procedure are:

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

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

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

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

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

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

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

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

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

```
ACS_TRANSIENT, ACS_PERMANENT,
```

otherwise (these latter two codes are discussed in Table 2.1 on page 29 of *Volume One*). The routine `AcAssocResponse` is then called to deal with the incoming association. The arguments are strictly boilerplate: they will work unaltered for most applications. If the association was accepted, then the routine `RoSetService` is used to tell the remote operations library to use the presentation service as the underlying service.

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

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

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

```
/* ryresponder.c - generic idempotent responder */

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

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

/* DATA */

int     debug = 0;

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";


static jmp_buf toplevel;


static IFP      startfnx;
static IFP      stopfnx;

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


extern int  errno;

/* RESPONDER */

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

```
        stop;
{
    register struct dispatch   *ds;
    AEI     aei;
    struct TSAPdisconnect   tds;
    struct TSAPdisconnect   *td = &tds;
    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);
    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);
        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);

    for (ds = dispatches; ds -> ds_name; ds++)
        if (RyDispatch (NOTOK, ops, ds -> ds_operation, ds -> ds_vector, roi)
                == NOTOK)
```

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

    startfnx = start;
    stopfnx = stop;

    if (isodeserver (argc, argv, aei, ros_init, ros_work, ros_lose, td)
            == NOTOK) {
        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;
}


static int  ros_init (vecp, vec)
int     vecp;
char  **vec;
{
    int     reply,
            result,
            sd;
    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;

    if (AcInit (vecp, vec, acs, aci) == NOTOK) {
        acs_advise (aca, "initialization fails");
```

```
        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);

    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,
                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;
    }
    if (reply != ACS_ACCEPT)
        return NOTOK;

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

    return sd;
}
```

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

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

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

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

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

    return OK;
}
```

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

    switch (roi -> roi_type) {
        case ROI_INVOKE:
        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)
                    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;

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

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

```
            break;

        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);

                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
                    if (reply != ACS_ACCEPT)
                        break;
                longjmp (toplevel, DONE);
            }
        /* NOTREACHED */

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


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

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

...
```

10.1.2 Operation Response

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

```
...

static int  op_name (sd, ryo, rox, in, roi)
int         sd;
struct RyOperation *ryo;
struct RoSAPinvoke *rox;
caddr_t   in;
struct RoSAPindication *roi;
{
    int         result,
                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);
    }
    if (debug)
        advise (LLOG_DEBUG, NULLCP, "RO-INVOKE.INDICATION/%d:  %s",
                sd, ryo -> ryo_name);
```

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

    if (won) {
            register struct type_MODULE_result *res;

/* 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 {
            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 */

    return result;
}

...

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");

    return OK;
}


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;
}

...
```

10.1.3 Error Handling

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

> `ros_adios`: used to report a ROS error and terminate;
>
> `ros_advise`: used to report a ROS error;
>
> `acs_advise`: used to report an ACS error;
>
> `adios`: used to report an error and terminate;
>
> `advise`: used to report an error; and,

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

```
...

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

   longjmp (toplevel, NOTOK);
}


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

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

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


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

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

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


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

    va_start (ap);

    _ll_log (pgm_log, LLOG_FATAL, ap);

    va_end (ap);

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

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


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

    va_start (ap);

    code = va_arg (ap, int);

    _ll_log (pgm_log, code, ap);

    va_end (ap);
}
#else
/* VARARGS3 */

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

10.1.4 An Example

An example of a responder written using this boilerplate is shown in Section 11.4 on page 184.

Chapter 11

A Complete Example

Now, let's consider a complete example. Our application is modeled after the example given in [Coo82]. 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
```

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 173.

The *Makefile* for the service is shown in Figure 11.2 starting on page 177.

```
PasswordLookup DEFINITIONS ::=

BEGIN

-- operations

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

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

-- errors

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

                              -- congestion at responder
congested ERROR
          ::=         1

-- types

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

```
        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
```

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

END
```

Figure 11.1: Remote Operations Module for the Password Lookup Service

```
#    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)
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


# Here it is...

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


# lookupd

inst-lookupd:   $(SBINDIR)ros.lookup

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

```
		-@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
```

```
$(BINDIR)lookup:        lookup
                -cp $@ zlookup
                -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 \
                        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

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
```

```
# clean

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

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:;
```

Figure 11.2: Makefile for the Password Lookup Service

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 183.

```
/* 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";

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 ();

                                        /* 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",

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

```
    "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;
```

```
{
    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;
}
```

```
/*  */

/* ARGSUSED */

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

/*  */

/* ARGSUSED */

static int  do_quit (sd, ds, args, dummy)
int        sd;
struct dispatch *ds;
char  **args;
caddr_t *dummy;
{
    struct AcSAPrelease acrs;
    register struct AcSAPrelease   *acr = &acrs;
    struct AcSAPindication  acis;
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort *aca = &aci -> aci_abort;

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

    if (!acr -> acr_affirmative) {
        (void) AcUAbortRequest (sd, NULLPEP, 0, aci);
```

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

    ACRFREE (acr);

    exit (0);
}

/* RESULTS */

/* ARGSUSED */

static int  lookup_result (sd, id, dummy, result, roi)
int     sd,
        id,
        dummy;
register struct type_PasswordLookup_Passwd *result;
struct RoSAPindication *roi;
{
    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');

    return OK;
}


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

```
    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 */

static int  lookup_error (sd, id, error, parameter, roi)
int     sd,
        id,
        error;
caddr_t parameter;
struct RoSAPindication *roi;
{
    register struct RyError *rye;

    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);

    return OK;
}
```

Figure 11.3: Initiator for the Password Lookup Service

11.4 Responder

The responder for the service is placed in *lookupd.c*. This file is shown in Figure 11.4 starting on page 189. Note that if actual network addresses were associated with the service definitions made earlier, then the responder could operate using the static discipline.

```c
/* 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) \
        ((p) = (type) calloc (1, sizeof *(p)))

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


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


/* DATA */

static char *myservice = "passwdstore";

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


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

static struct dispatch dispatches[] = {
```

```
    "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;
```

```
    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)
```

```
        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;
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
                || (*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
                        && (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 {
```

```
            res -> uid -> arg = pw -> pw_uid;
            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);
    }
    else
        result = error (sd, error_PasswordLookup_noSuchUser, (caddr_t) NULL,
                rox, roi);

    return result;
}


/* ERROR */

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");

    return OK;
}


/* U-REJECT */

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;
}
```

Figure 11.4: Responder for the Password Lookup Service

Bibliography

[CCI88a] Remote Operations: Model, Notation and Service Definition. International Telegraph and Telephone Consultative Committee, March 1988. Recommendation X.219.

[CCI88b] Remote Operations: Protocol Specification. International Telegraph and Telephone Consultative Committee, March 1988. Recommendation X.229.

[CCI88c] Specification of Abstract Syntax Notation One. International Telegraph and Telephone Consultative Committee, 1988. Recommendation X.208.

[Coo82] Eric C. Cooper. Writing Distributed Programs with Courier. Technical report, Computer Science Division — EECS, University of California, Berkeley, March 1982.

[ECM85] Remote Operations: Concepts, Notation and Connection-Oriented Mappings, December 1985. ECMA TR/31.

[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.

[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.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[Pos81] Jon B. Postel. Transmission Control Protocol. Request for Comments 793, DDN Network Information Center, SRI International, September 1981. See also MIL-STD 1778.

[RC86] 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.

[Ros90] Marshall T. Rose. *The Open Book: A Practical Perspective on Open Systems Interconnection*. Prentice-hall, 1990. ISBN 0–13–643016–3.

Index