The ISO Development Environment: User's Manual

Volume 1: Application Services

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 #6.24) was LaTeXset February 22, 1994 with lplain v2.09-March 25, 1992.

List of Tables

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

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 #6.24) 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

Application Services

Chapter 2

Association Control

The *libacsap*(3n) library implements the Association Control Service (ACS). Logically, one views an application to consist of several *application service elements* (ASE's). Although these ASEs cooperate, each performs a different function for the application. The Association Control Service Element (ACSE) is concerned with the task of "starting" and "stopping" the network for the application. That is, an application uses the ACSE to establish a connection, termed an *association*, between two users. The association *binds* the two users, which are referred to as the *initiator* and the *responder*. This association, once established is used by other ASEs (e.g., the remote operations service element). Later, the ACSE is called upon to release the association, either gracefully (perhaps with some negotiation), or abruptly (with possible loss of information).

Like most models of OSI services, the underlying assumption is one of an asynchronous environment: the service provider may generate events for the service user without the latter entity triggering the actions which led to the event. For example, in a synchronous environment, an indication that data has arrived usually occurs only when the service user asks the service provider to read data; in an asynchronous environment, the service provider may interrupt the service user at any time to announce the arrival of data.

The `acsap` module in this release presents a synchronous interface. However once the association is established, an asynchronous interface may be selected. The `acsap` module itself is naive as to the interface being used: the particular application service element which is responsible for transferring data will manage the association once the ACSE has established it.

All of the routines in the *libacsap*(3n) library are integer-valued. They return the manifest constant `OK` on success, or `NOTOK` otherwise. In some circumstances, failures are fatal and cause (or are caused by) the association being released. Section 2.7 describes how such failures may be distinguished.

2.1 An Important Note

In the current release there are several ways to establish an association. This is due to recent changes in the OSI application layer, and a desire to remain compatible with previous work. Users are strongly encouraged to use the new facilities described herein, as the older facilities will eventually be removed.

Here are the new facilities available:

Desired Service	Association Primitives	Section(s)
Remote Operations (complete discipline) [ISO88e, CCI88c]	A-ASSOCIATE A-RELEASE A-ABORT	2.2 3.3.1
Remote Operations (complete discipline with reliable transfer) [ISO88e, CCI88c]	RT-OPEN RT-CLOSE	4.1 3.3.1
Reliable Transfer [ISO88c, CCI88a]	RT-OPEN RT-CLOSE	4.1

Here are the old facilities available:

Desired Service	Association Primitives	Section
Remote Operations (basic discipline) [ECM85]	RO-BEGIN RO-END	A.1
Remote Operations (advanced discipline with reliable transfer) [CCI84b]	X.410 OPEN X.410 CLOSE	A.2 3.3.1
Reliable Transfer [CCI84b]	X.410 OPEN X.410 CLOSE	A.2

In short, depending on whether a Reliable Transfer Service Element (RTSE) is desired, for the purposes of association control, all new applications should use either the library discussed below, or the reliable transfer routines discussed in Section 4.1 on page 87.

2.2 Associations

There are three aspects of association management: *association establishment*, *association release*, and, *association abort*. Each of these are now described in turn.

2.2.1 Association Establishment

The *libacsap*(3n) library distinguishes between the user which started an association, the *initiator*, and the user which was subsequently bound to the association, the *responder*. We sometimes term these two entities the *client* and the *server*, respectively.

Addresses

Addresses for the association control service entity consist of two parts: a presentation address (as discussed in Section 2.2 on page 16 of *Volume Two*), and application-entity information. Internally, the `AEInfo` is used to represent this notion:

```
typedef struct AEInfo {
    PE      aei_ap_title;
    PE      aei_ae_qualifier;

    int     aei_ap_id;
    int     aei_ae_id;

    int     aei_flags;
#define AEI_NULL        0x00
#define AEI_AP_ID       0x01
#define AEI_AE_ID       0x02
}       AEInfo, *AEI;
#define NULLAEI         ((AEI) 0)
```

This structure contains several elements:

`aei_ap_title`: the application-process title;

`aei_ae_qualifier`: the application-entity qualifier;

`aei_ap_id`: the application-process invocation identifier;

`aei_ae_id`: the application-entity invocation identifier; and,

`aei_flags`: flags, indicating which, if any, of the invocation-identifiers are present.

The routing `sprintaei` can be used to return a null-terminated string describing the application-entity.

```
char   *sprintaei (aei)
AEI     aei;
```

Finally, the routine `oid2aei` converts object identifiers (discussed in Section 5.4.6 on page 135) and converts them to application-entity information. This is used by the stub-directory service:

```
AEI     oid2aei (oid)
OID     oid;
```

In Figure 2.1, an example of how one constructs the address for the File Transfer, Access and Management (FTAM) service on host `RemoteHost` is presented. The key routine is `_str2aei`:

```
AEI     _str2aei (designator, qualifier, context, interactive)
char   *designator,
       *qualifier,
       *context;
int     interactive;
```

The parameters to this procedure are:

`designator`: input from the user (e.g., “`hubris, cs, ucl, gb`” or “`hubris`”);

`qualifier`: the service qualifier (e.g., `filestore`);

`context`: the `supportedApplicationContext` for the desired service (e.g., `iso ftam`); and;

`interactive`: non-zero if the user’s terminal can be queried for additional information.

The routine `_str2aei` will first attempt to resolution using the “user-friendly nameservice” with the `designator`, `context`, and `interactive` parameters. If this fails, the “stub directory service” will be used, with the `designator` and `qualifier` parameters.

For backwards-compatiblity, a macro, `str2aei` is provided which is equivalent to:

```
_str2aei (designator, qualifier, NULLCP, 0)
```

For future-compability, you should start using the `str2aeinfo` routine, as both `_str2aei` and its macro-ized counterpart will soon be deprecated:

```
AEI     str2aeinfo (string, context, interactive, userdn, passwd)
char   *string,
       *context,
       *userdn,
       *passwd;
int     interactive;
```

The parameters to this procedure are:

`string`: input from the user (e.g., “`hubris, cs, ucl, gb`” or “`hubris`”);

`context`: the `supportedApplicationContext` for the desired service (e.g., `iso ftam`);

`interactive`: non-zero if the user's terminal can be queried for additional information;

`userdn`: the Distinguished Name (DN) when using the Directory service; and;

`passwd`: authentication information for the Directory service.

This routine attempts resolution solely via the "user-friendly nameservice".

The routine `aei2addr` takes application-entity information, and returns the appropriate presentation address.

```
struct PSAPaddr *aei2addr (aei)
AEI     aei;
```

Address Encodings

It may be useful to encode a presentation address for viewing. Although a consensus for a standard way of doing this has not yet been reached, the routines `paddr2str` and `str2paddr` may be used in the interim. These implement the encodings defined in [Kil89]. The BNF syntax for this encoding is shown in Figure 2.2 on page 23.

```
char   *paddr2str (pa, na)
struct PSAPaddr *pa;
struct NSAPaddr *na;
```

The parameters to this procedure are:

`pa`: the presentation address; and,

`na`: a network address to use instead of the network addresses in the `pa` parameter (use the manifest constant `NULLNA` otherwise).

If `paddr2str` fails, it returns the manifest constant `NULLCP`.

The routine `str2paddr` takes an ascii string encoding and returns a presentation address.

```
#include <isode/acsap.h>

...

register struct PSAPaddr *pa;
register AEI aei;

...

char *qualifier = "filestore";
char *context = "iso ftam";

if ((aei = _str2aei ("hubris, cs, ucl, gb", qualifier,
                        context, isatty (fileno (stdin)),
                        NULLCP, NULLCP))
        == NULLAEI)
    error ("unable to resolve service:  %s", PY_pepy);
if ((pa = aei2addr (aei)) == NULLPA)
    error ("address translation failed");

...
```

Figure 2.1: Constructing the address for the FTAM entity

```
struct PSAPaddr *str2paddr (str)
char   *str;
```

The parameter to this procedure is:

`str`: the ascii string.

If `str2paddr` fails, it returns the manifest constant `NULLPA`.

`paddr2str` is really a macro which calls the routine `_paddr2str`:

```
char  *_paddr2str (pa, na, compact)
struct PSAPaddr *pa;
struct NSAPaddr *na;
int    compact;
```

The parameters to this procedure are:

`pa`: the presentation address;

`na`: a network address to use instead of the network addresses in the `pa` parameter (use the manifest constant `NULLNA` otherwise); and,

`compact`: indicates the style of encoding.

If `compact` is greater than zero, then the encoding reflects a normalized network address. This is useful for passing to arbitrary processes in the network. If `compact` is less than zero, then the encoding reflects the BNF in Figure 2.2, without any macro substitutions. If `compact` is equal to zero (which is what `paddr2str` uses), then the encoding reflects the above BNF with macro substitutions.

The routine `pa2str` takes a presentation address and returns a null-terminated ascii string suitable for viewing:

```
char   *pa2str (pa)
struct PSAPaddr *pa;
```

The parameter to this procedure is:

`pa` : the presentation address to be printed.

```
<presentation-address> ::=
                 [[[ <psel> "/" ] <ssel> "/" ] <tsel> "/" ]
                 <network-address-list>

<network-address-list> ::=
                 <network-address> "|" <network-address-list>
             | <network-address>

<psel> ::=       <selector>
<ssel> ::=       <selector>
<tsel> ::=       <selector>

<selector> ::=
                 '"' <otherstring> '"'        -- IA5
                                              -- (for other chars use hex)
             | "#" <digitstring>              -- US GOSIP
             | "'" <hexstring> "'H"           -- hex
             | ""                             -- empty but present

<network-address> ::=
                 "NS" "+" <hexstring>         -- concrete binary
                                              -- this is the compact
                                              --   encoding
             | <afi> "+" <idi> [ "+" <dsp> ]


<afi>            ::= "X121" | "DCC" | "TELEX" | "PSTN" | "ISDN"
                         | "ICD" | "LOCAL"

<idi>            ::= <digitstring>

<dsp>  ::=       "d" <digitstring>            -- abstract decimal
             | "x" <hexstring>                -- abstract binary
             | "l" <otherstring>              -- IA5: local form only
             | "RFC-1006" "+" <prefix> "+" <ip>
                         [ "+" <port> [ "+" <tset> ]]
             | "X.25(80)" "+" <prefix> "+" <dte>
                         [ "+" <cudf-or-pid> "+" <hexstring> ]
             | "ECMA-117-Binary" "+" <hexstring> "+" <hexstring>
                         "+" <hexstring>
             | "ECMA-117-Decimal" "+" <digitstring> "+" <digitstring>
                         "+" <digitstring>
```

Figure 2.2: BNF Syntax for paddr2str/str2paddr

```
<prefix>        ::= <digit> <digit>

<ip>            ::= <otherstring>
                        -- dotted decimal form (e.g., 10.0.0.6)
                        -- or domain (e.g., twg.com)
<port>          ::= <digitstring>
<tset>          ::= <digitstring>

<dte>           ::= <digitstring>
<cudf-or-pid>   ::= "CUDF" | "PID"

<digitstring>   ::= <digit>  <digitstring> | <digit>
<digit>         ::= [0-9]

<otherstring>   ::= <other>  <otherstring> | <other>
<other>         ::= [0-9a-zA-Z+-.]

<hexstring>     ::= <hexdigit> <hexstring>  | <hexdigit>
<hexdigit>      ::= [0-9a-f]
```

Figure 2.2: BNF Syntax for paddr2str/str2paddr (continued)

Server Initialization

The *tsapd*(8c) daemon, upon accepting a connection from an initiating host, consults the ISO services database to determine which program on the local system implements the desired application context.

Once the program has been ascertained, the daemon runs the program with any arguments listed in the database. In addition, it appends some *magic arguments* to the argument vector. Hence, the very first action performed by the responder is to re-capture the ACSE state contained in the magic arguments. This is done by calling the routine `AcInit`, which on a successful return, is equivalent to a A-ASSOCIATE.INDICATION event from the association control service provider.

```
int     AcInit (vecp, vec, acs, aci)
int     vecp;
char  **vec;
struct AcSAPstart *acs;
struct AcSAPindication *aci;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

`acs`: a pointer to an `AcSAPstart` structure, which is updated only if the call succeeds; and,

`aci`: a pointer to an `AcSAPindication` structure, which is updated only if the call fails.

If `AcInit` is successful, it returns information in the `acs` parameter, which is a pointer to an `AcSAPstart` structure.

```
struct AcSAPstart {
    int     acs_sd;

    OID     acs_context;

    AEInfo  acs_callingtitle;
    AEInfo  acs_calledtitle;
```

```
    struct PSAPstart acs_start;

    int     acs_ninfo;
    PE      acs_info[NACDATA];
};
```

The elements of this structure are:

`acs_sd`: the association-descriptor to be used to reference this association;

`acs_context`: the application context name;

`acs_callingtitle`: information on the calling application-entity (if any);

`acs_calledtitle`: information on the called application-entity (if any);

`acs_start`: an `PSAPstart` structure (consult page 20 of *Volume Two*); and,

`acs_info/acs_ninfo`: any initial data (and the length of that data).

Note that the proposed contexts list in the `acs_start` element will contain a context for the ACSE. The routine `AcFindPCI` can be used to determine the PCI being used by the ACSE:

```
int     AcFindPCI (sd, pci, aci)
int     sd;
int    *pci;
struct AcSAPindication *aci;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`pci`: a pointer to an integer location, which is updated only if the call succeeds; and,

`aci`: a pointer to an `AcSAPindication` structure, which is updated only if the call fails.

Further, note that the data contained in the structure was allocated via *malloc*(3), and should be released by using the `ACSFREE` macro when no longer referenced. The `ACSFREE` macro, behaves as if it was defined as:

```
void    ACSFREE (acs)
struct AcSAPstart *acs;
```

The macro frees only the data allocated by `AcInit`, and not the `AcSAPstart` structure itself. Further, `ACSFREE` should be called only if the call to the `AcInit` routine returned `OK`.

If the call to `AcInit` is not successful, then a A-P-ABORT.INDICATION event is simulated, and the relevant information is returned in an encoded `AcSAPindication` structure.

```
struct AcSAPindication {
    int     aci_type;
#define ACI_FINISH      0x00
#define ACI_ABORT       0x01

    union {
        struct AcSAPfinish aci_un_finish;
        struct AcSAPabort aci_un_abort;
    }   aci_un;
#define aci_finish      aci_un.aci_un_finish
#define aci_abort       aci_un.aci_un_abort
};
```

As shown, this structure is really a discriminated union (a structure with a tag element followed by a union). Hence, on a failure return, one first coerces a pointer to the `AcSAPabort` structure contained therein, and then consults the elements of that structure.

```
struct AcSAPabort {
    int     aca_source;

    int     aca_reason;

    int     aca_ninfo;
    PE      aca_info[NACDATA];
```

```
#define ACA_SIZE        512
    int     aca_cc;
    char    aca_data[ACA_SIZE];
};
```

The elements of an AcSAPabort structure are:

aca_source: the source of the abort, one of:

Value	Source
ACA_USER	service-user (peer)
ACA_PROVIDER	service-provider
ACA_LOCAL	local ACPM

aca_reason: the reason for the provider-initiated abort (meaningful only if aca_source is not ACA_REQUESTOR), consult Table 2.1;

aca_info/aca_ninfo: any abort data (and the length of that data) from the peer (if aca_source is ACA_USER); and,

aca_data/aca_cc: a diagnostic string from the provider.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released by using the ACAFREE macro when no longer referenced. The ACAFREE macro, behaves as if it was defined as:

```
void    ACAFREE (aca)
struct AcSAPabort *aca;
```

The macro frees only the data contained in the structure, and not the AcSAPabort structure itself.

After examining the information returned by AcInit on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the association. For either response, the responder should use the AcAssocResponse routine (which corresponds to the A-ASSOCIATE.RESPONSE action).

```
int     AcAssocResponse (sd, status, reason, context,
        respondtitle, respondaddr, ctxlist, defctxresult,
        prequirements, srequirements, isn, settings, ref,
```

Provider-Initiated Aborts (fatal)	
`ACS_ADDRESS`	Address unknown
`ACS_REFUSED`	Connect request refused on this network connection
`ACS_CONGEST`	Local limit exceeded
`ACS_PRESENTATION`	Presentation disconnect
`ACS_PROTOCOL`	Protocol error
`ACS_RESPONDING`	Rejected by responding ACPM
`ACS_ABORTED`	Peer aborted association
User-Initiated Rejections (fatal)	
`ACS_PERMANENT`	Permanent
`ACS_TRANSIENT`	Transient
User-Initiated Rejections (non-fatal)	
`ACS_REJECT`	Release rejected
Interface Errors (non-fatal)	
`ACS_PARAMETER`	Invalid parameter
`ACS_OPERATION`	Invalid operation

Table 2.1: AcSAP Failure Codes

```
        data, ndata, aci)
int     sd;
int     status,
        reason;
OID     context;
AEI     respondtitle;
struct PSAPaddr *respondaddr;
int     prequirements,
        srequirements,
        settings,
        ndata;
long    isn;
struct PSAPctxlist *ctxlist;
int     defctxresult;
struct SSAPref *ref;
PE     *data;
struct AcSAPindication *aci;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`status`: the acceptance indicator (either `ACS_ACCEPT` if the association is to be accepted, or one of the user-initiated rejection codes listed in Table 2.1 on page 29);

`reason`: the diagnostic (either `ACS_USER_NULL` if the association is to be accepted, or one of the user-initiated diagnostic codes listed in Table 2.2 on page 31);

`context`: the application context to be used over the association (defaults to the context proposed);

`respondtitle`: information on the responding application-entity (optional);

`data/ndata`: an array of user data (and the number of elements in that array, consult the warning on page 49); and,

`aci`: a pointer to an `AcSAPindication` structure, which is updated only if the call fails.

Provider-Initiated Diagnostics	
`ACS_PROV_NULL`	Null
`ACS_PROV_NOREASON`	No reason given
`ACS_VERSION`	No common acse version
User-Initiated Diagnostics	
`ACS_USER_NULL`	Null
`ACS_USER_NOREASON`	No reason given
`ACS_CONTEXT`	Application context name not supporte
`ACS_CALLING_AP_TITLE`	Calling AP title not recognized
`ACS_CALLING_AP_ID`	Calling AP invocation-ID not recognized
`ACS_CALLING_AE_QUAL`	Calling AE qualifier not recognized
`ACS_CALLING_AE_ID`	Calling AE invocation-ID not recognized
`ACS_CALLED_AP_TITLE`	Called AP title not recognized
`ACS_CALLED_AP_ID`	Called AP invocation-ID not recognized
`ACS_CALLED_AE_QUAL`	Called AE qualifier not recognized
`ACS_CALLED_AE_ID`	Called AE invocation-ID not recognized

Table 2.2: AcSAP Diagnostic Codes

The remaining parameters are for the presentation service, consult the description of the `PConnResponse` routine on page 27 of *Volume Two*.

If the call to `AcAssocResponse` is successful, and the `status` parameter is set to `ACS_ACCEPT`, then association establishment has now been completed. If the call is successful, but the `status` parameter is not `ACS_ACCEPT`, then the association has been rejected and the responder may exit. Otherwise, if the call failed and the reason is "fatal", then the association is lost.

Client Initialization

A program wishing to associate itself with another application-level user calls the `AcAssocRequest` routine, which corresponds to the user taking the A-ASSOCIATE.REQUEST action.

```
int     AcAssocRequest (context, callingtitle, calledtitle,
        callingaddr, calledaddr, ctxlist, defctxname,
```

```
        prequirements, srequirements, isn, settings, ref,
        data, ndata, qos, acc, aci)
OID     context;
AEI     callingtitle,
        calledtitle;
struct PSAPaddr *callingaddr,
                *calledaddr;
int     prequirements,
        srequirements,
        settings,
        ndata;
long    isn;
struct PSAPctxlist *ctxlist;
OID     defctxname;
struct SSAPref *ref;
PE    *data;
struct QOStype *qos;
struct AcSAPconnect *acc;
struct AcSAPindication *aci;
```

The parameters to this procedure are:

`context`: the application context to be used over the association;

`callingtitle`: information on the calling application-entity (use the manifest constant `NULLAEI` if not specified);

`calledtitle`: information on the called application-entity (use the manifest constant `NULLAEI` if not specified);

`data/ndata`: an array of initial data (and the number of elements in that array, consult the warning on page 49);

`qos`: the quality of service on the connection (see Section 4.6.2 in *Volume Two*);

`acc`: a pointer to an `AcSAPconnect` structure, which is updated only if the call succeeds; and,

`aci`: a pointer to an `AcSAPindication` structure, which is updated only if the call fails.

The remaining parameters are for the presentation service, consult the description of the PConnRequest routine on page 29 of *Volume Two* for additional information. Note that the ctxlist parameter is mandatory: the association control service element will look for the ACSE PCI there. If not present, it will add the PCI to the list.

If the call to AcAssocRequest is successful (the A-ASSOCIATE.CONFIRMATION event occurs), it returns information in the acc parameter which is a pointer to an AcSAPconnect structure.

```
struct AcSAPconnect {
    int     acc_sd;

    int     acc_result;
    int     acc_diagnostic;

    OID     acc_context;

    AEInfo  acc_respondtitle;

    struct PSAPconnect acc_connect;

    int     acc_ninfo;
    PE      acc_info[NACDATA];
};
```

The elements of an AcSAPconnect structure are:

acc_sd: the association-descriptor to be used to reference this association;

acc_result/acc_diagnostic: the association response and diagnostic;

acc_context: the application context name;

acc_respondtitle: information on the responding application-entity (if any);

acc_connect: an PSAPconnect structure (consult page 30 of *Volume Two*); and,

acc_info/acc_ninfo: any initial data (and the length of that data).

Note that the negotiated contexts list in the `acc_connect` element will contain a context for the ACSE. To determine the PCI for the ACSE context, the routine `AcFindPCI` routine, described above can be used.

If the call to `AcAssocRequest` is successful, and the `acc_result` element is set to `ACC_ACCEPT`, then association establishment has completed. If the call is successful, but the `acc_result` element is not `ACC_ACCEPT`, then the association attempt has been rejected; consult Table 2.1 on page 29 and Table 2.2 on page 31 to determine the reason for the rejection (further information can be found in the `aci` parameter). Otherwise, if the call fails then the association is not established and the `AcSAPabort` structure of the `AsCAPindication` discriminated union has been updated.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released by using the `ACCFREE` macro when no longer referenced. The `ACCFREE` macro, behaves as if it was defined as:

```
void    ACCFREE (acc)
struct AcSAPconnect *acc;
```

The macro frees only the data allocated by `AcAssocRequest`, and not the `AcSAPconnect` structure itself. Further, `ACCFREE` should be called only if the call to the `AcAssocRequest` routine returned `OK`.

Normally `AcAssocRequest` returns only after an association has succeeded or failed. This is termed a *synchronous* association initiation. If the user desires, an *asynchronous* association may be initiated. This routine is really a macro which calls the routine `AcAsynAssocRequest` with an argument indicating that a association should be attempted synchronously.

```
int     AcAsynAssocRequest (context, callingtitle, calledtitle,
        callingaddr, calledaddr, ctxlist, defctxname,
        prequirements, srequirements, isn, settings, ref,
        data, ndata, qos, acc, aci, async)
OID     context;
AEI     callingtitle,
        calledtitle;
struct PSAPaddr *callingaddr,
                *calledaddr;
int     prequirements,
        srequirements,
        settings,
```

```
        ndata,
        async;
long    isn;
struct PSAPctxlist *ctxlist;
OID     defctxname;
struct SSAPref *ref;
PE    *data;
struct QOStype *qos;
struct AcSAPconnect *acc;
struct AcSAPindication *aci;
```

The additional parameter to this procedure is:

`async`: whether the association should be initiated asynchronously.

If the `async` parameter is non-zero, then `AcAsynAssocRequest` returns one of four values: `NOTOK`, which indicates that the association request failed; `DONE`, which indicates that the association request succeeded; or, either of `CONNECTING_1` or `CONNECTING_2`, which indicates that the connection request is still in progress. In the first two cases, the usual procedures for handling return values from `AcAssocRequest` are employed (i.e., a `NOTOK` return from `AcAsynAssocRequest` is equivalent to a `NOTOK` return from `AcAssocRequest`, and, a `DONE` return from `AcAsynAssocRequest` is equivalent to a `OK` return from `AcAssocRequest`).

In the final case, when either `CONNECTING_1` or `CONNECTING_2` is returned, only the `acc_sd` element of the `acc` parameter has been updated; it reflects the association-descriptor to be used to reference this association. To determine when the association attempt has been completed, the routine `xselect` (consult Section 2.4) should be used after calling `PSelectMask`. In order to determine if the association attempt is successful, the routine `AcAsynRetryRequest` is called:

```
int     AcAsynRetryRequest (sd, acc, aci)
int     sd;
struct AcSAPconnect *acc;
struct AcSAPindication  *aci;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`acc`: a pointer to an `AcSAPconnect` structure, which is updated only if the call succeeds; and,

`aci`: a pointer to an `AcSAPindication` structure, which is updated only if the call fails.

Again, one of three values are returned: `NOTOK`, which indicates that the association request failed; `DONE`, which indicates that the association request succeeded; and, `CONNECTING_1` or `CONNECTING_2` which indicates that the connection request is still in progress.

Refer to Section 4.2.3 on page 112 in *Volume Two* for information on how to make efficient use of the asynchronous connection facility.

2.2.2 Association Release

The `AcRelRequest` routine is used to request the release of an association, and corresponds to a A-RELEASE.REQUEST action.

```
int     AcRelRequest (sd, reason, data, ndata, secs, acr, aci)
int     sd;
int     reason;
PE     *data;
int     ndata;
int     secs;
struct AcSAPrelease *acr;
struct AcSAPindication *aci;
```

The parameters to this procedure:

`sd`: the association-descriptor;

`reason`: the reason why the association should be released, one of:

Value	Reason
ACF_NORMAL	normal
ACF_URGENT	urgent
ACF_UNDEFINED	undefined

`data/ndata`: an array of final data (and the number of elements in that array, consult the warning on page 49);

`secs`: the maximum number of seconds to wait for a response (use the manifest constant `NOTOK` if no time-out is desired);

`acr`: a pointer to an `AcSAPrelease` structure, which is updated only if the call succeeds; and,

`aci`: a pointer to an `AcSAPindication` structure, which is updated only if the call fails.

If the call to `AcRelRequest` is successful, then this corresponds to a A-RELEASE.CONFIRMATION event, and it returns information in the `acr` parameter, which is a pointer to an `AcSAPrelease` structure.

```
struct AcSAPrelease {
    int     acr_affirmative;

    int     acr_reason;

    int     acr_ninfo;
    PE      acr_info[NACDATA];
};
```

The elements of this structure are:

`acr_affirmative`: the acceptance indicator;

`acr_reason`: the reason for the indicator, one of:

Value	Reason
ACR_NORMAL	normal
ACR_NOTFINISHED	not finished
ACR_UNDEFINED	undefined

`acr_info/acr_ninfo`: any final data (and the length of that data).

Note that the data contained in the structure was allocated via *malloc*(3), and should be released by using the `ACRFREE` macro when no longer referenced. The `ACRFREE` macro, behaves as if it was defined as:

```
void    ACRFREE (acr)
struct AcSAPrelease *acr;
```

The macro frees only the data allocated by AcRelRequest, and not the AcSAPrelease structure itself. Further, ACRFREE should be called only if the call to the AcRelRequest routine returned OK.

If the call to AcRelRequest is successful, and the acr_affirmative element is set, then the association has been released. If the call is successful, but the acr_affirmative element is not set (i.e., zero), then the request to release the association has been rejected by the remote user, and the association is still established. Otherwise the AcSAPabort element of the AcSAPindication parameter aci contains the reason for failure.

Note that if a non-negative value is given to the secs parameter and a response is not received within this number of seconds, then the value contained in the aca_reason element is ACS_TIMER. The user can then call the routine AcRelRetryRequest to continue waiting for a response:

```
int     AcRelRetryRequest (sd, secs, acr, aci)
int     sd;
int     secs;
struct AcSAPrelease  *acr;
struct AcSAPindication  *aci;
```

The parameters to this procedure are:

sd: the association-descriptor;

secs: the maximum number of seconds to wait for a response (use the manifest constant NOTOK if no time-out is desired);

acr: a pointer to a AcSAPrelease structure, which is updated only if the call succeeds; and,

aci: a pointer to a AcSAPindication structure, which is updated only if the call fails.

If the call to AcRelRetryRequest is successful, and the acr_affirmative element is set, then the association has been closed. If the call is successful, but the acr_affirmative element is not set (i.e., zero), then the request to release the assocation has been rejected by the remote user, and the association is still open. Otherwise the AcSAPabort structure contained in the `AcSAPindication` parameter `aci` contains the reason for failure. As expected, the value `ACS_TIMER` indicates that no response was received within the time given by the `secs` parameter.

Upon receiving a A-RELEASE.INDICATION event, the user is required to generate a A-RELEASE.RESPONSE action using the `AcRelResponse` routine.

```
int     AcRelResponse (sd, status, reason, data, ndata, aci)
int     sd;
int     status,
        reason;
PE     *data;
int     ndata;
struct AcSAPindication *aci;
```

The parameters to this procedure:

`sd`: the association-descriptor;

`status`: the acceptance indicator (either `ACS_ACCEPT` if the association is to be released, or `ACS_REJECT` otherwise);

`reason`: the reason for the indicator, one of:

Value	**Reason**
ACR_NORMAL	normal
ACR_NOTFINISHED	not finished
ACR_UNDEFINED	undefined

`data/ndata`: an array of final data (and the number of elements in that array, consult the warning on page 49); and,

`aci`: a pointer to an `AcSAPindication` structure, which is updated only if the call fails.

If the call to `AcRelResponse` is successful, and if the `result` parameter is set to `ACS_ACCEPT`, then the association has been released. If the call is successful, but the `result` parameter is not `ACS_ACCEPT`, then the association remains established.

2.2.3 Association Abort

To unilaterally abort an association, the routine `AcUAbortRequest` routine is used which corresponds to the A-ABORT.REQUEST action.

```
int     AcUAbortRequest (sd, data, ndata, aci)
int     sd;
PE     *data;
int     ndata;
struct AcSAPindication *aci;
```

The parameters to this procedure:

`sd`: the association-descriptor;

`data/ndata`: an array of abort data (and the number of elements in that array, consult the warning on page 49); and,

`aci`: a pointer to an `AcSAPindication` structure, which is updated only if the call fails.

If the call to `AcUAbortRequest` is successful, then the association is immediately released, and any data queued for the association may be lost.

2.3 Association Events

Typically, the presentation service generates events which lead to the A-RELEASE.INDICATION, A-ABORT.INDICATION, and A-P-ABORT.INDICATION events being raised. For those interested in writing application service elements which interpret this information, this section describes the necessary mappings.

2.3.1 Release Indication

Upon receiving a P-RELEASE.INDICATION event, the routine `AcFINISHser` should be called to map this into a A-RELEASE.INDICATION event.

```
int     AcFINISHser (sd, pf, aci)
int     sd;
struct PSAPfinish *pf;
struct AcSAPindication *aci;
```

The parameters to this procedure:

`sd`: the association-descriptor;

`pa`: a pointer to an `PSAPfinish` structure, containing information on the presentation-level release; and,

`aci`: a pointer to an `AcSAPindication` structure.

If the call to `AcFINISHser` is successful, then the `aci_type` field of the `aci` parameter contains the value `ACI_FINISH`, and an `AcSAPfinish` structure is contained inside the `aci` parameter.

```
struct AcSAPfinish {
    int     acf_reason;

    int     acf_ninfo;
    char    acf_info[NACDATA];
};
```

The elements of this structure are:

`acf_reason`: the reason for the release request, one of:

Value	**Reason**
`ACF_NORMAL`	normal
`ACF_URGENT`	urgent
`ACF_UNDEFINED`	undefined

`acf_info/acf_ninfo`: any final data (and the length of that data).

Note that the data contained in the structure was allocated via *malloc*(3), and should be released by using the `ACFFREE` macro when no longer referenced. The `ACFFREE` macro, behaves as if it was defined as:

```
void    ACFFREE (acf)
struct AcSAPfinish *acf;
```

The macro frees only the data allocated by `AcFINISHser`, and not the `AcSAPfinish` structure itself. Further, `ACFFREE` should be called only if the call to the `AcFINISHser` routine returned `OK`.

Otherwise, if the call fails, the `aci` parameter contains information pertaining to the failure.

2.3.2 Abort Indication

Upon receiving a P-U-ABORT.INDICATION or P-P-ABORT.INDICATION event, the routine `AcABORTser` should be called to map this into a A-ABORT.INDICATION or A-P-ABORT.INDICATION event.

```
int     AcABORTser (sd, pa, aci)
int     sd;
struct PSAPabort *pa;
struct AcSAPindication *aci;
```

The parameters to this procedure:

`sd`: the association-descriptor;

`pa`: a pointer to an `PSAPabort` structure, containing information on the presentation-level abort; and,

`aci`: a pointer to an `AcSAPindication` structure.

If the call to `AcABORTser` is successful, then the `aci` parameter is updated to reflect information on the abort indication. Otherwise, if the call fails, the `aci` parameter contains information pertaining to the failure.

2.4 Select Facility

Ideally, the *select*(2) system call should be used on all descriptors, regardless of the level of abstraction (e.g., association-descriptor, presentation-descriptor, and so on). The routines with a `SelectMask` suffix, such as `PSelectMask` are supplied to determine if there are already queued events. These routines should always be called prior to using the select facility of the kernel. Sadly, some networking subsystems used by the software do not support *select*(2). To provide a consistent interface, the `xselect` routine should be used instead of *select*(2).

```
int     xselect (nfds, rfds, wfds, efds, secs)
int     nfds;
fd_set *rfds,
       *wfds,
       *efds,
        secs;
```

The parameters to this procedure are:

- `nfds`: the number of descriptors present in the masks (actually the `width` of descriptors from zero to the highest meaningful descriptor);
- `rfds/wfds/efds`: the locations of masks interested in read, write, and exception events; and,
- `secs`: the maximum number of seconds to wait for an event (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).

Unlike most routines, the `xselect` routine returns one of several values: `NOTOK` (on failure), `OK` (if no events occurred within the time limit), or, the number of descriptors on which events occurred (the masks are updated appropriately).

By default, unless the application explicitly ignores `SIGINT`, when the user types a CTRL-C this will interrupt the call to `xselect`. If you make calls directly to `xselect`, you can disable this behavior by setting

```
extern int xselect_blocking_on_intr;
```

to a non-zero value. (When reading from an association-descriptor, this behavior is automatically disabled.)

2.5 Generic Server Dispatch

Ideally, one should write a server which can operate in one of two modes:

- Each incoming connection results in *tsapd*(8c) invoking another instance of the server; or,
- All incoming connections are given to exactly one instance of the server (the server is invoked without arguments during system initialization).

The choice as to which mode is used is made by the system administrator. By default, the first mode, termed the *dynamic* approach, is used. The second mode, termed the *static* approach, is described in Section 10.2 on page 162.

There are several ways in which this dual-approach scheme can be realized. One such implementation is based on the routine `isodeserver`.

```
int     isodeserver (argc, argv, aei, initfnx, workfnx,
                losefnx, td)
int     argc;
char  **argv;
AEI     aei;
IFP     initfnx,
        workfnx,
        losefnx;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`argc`: the length of the argument vector which the program was invoked with;

`argv`: the argument vector which the program was invoked with;

`aei`: the information on the application-entity offering the service;

`initfnx`: the address of an event-handler routine to be invoked when a new connection arrives;

`workfnx`: the address of an event-handler routine to be invoked when activity occurs on an association;

`losefnx`: the address of an event-handler routine to be invoked if `TNetAccept` (described in Section 4.7) fails and,

`td`: a pointer to an `TSAPdisconnect` structure, which is updated only if the call fails.

If the call is successful, then the program may terminate immediately, as no work remains to be done (in the case of the single instance mode, `isodeserver` returns only on error). Otherwise, the `td` parameter indicates the reason for failure (consult Section 4.2.2 on page 108 of *Volume Two*).

When an event associated with a new connection occurs, the event-handler routine is invoked with two parameters:

```
(*initfnx) (vecp, vec);
int     vecp;
char  **vec;
```

The parameters are:

`vecp`: the length of the argument vector; and,

`vec`: the argument vector.

The event-handler should then call `AcInit` with these parameters to achieve an A-ASSOCIATION.INDICATION event. If `AcInit` is successful, the event-handler should decide if it wishes to honor the association and then call `AcAssocResponse` with the appropriate parameters. The event-handler should return the assocation-descriptor if it accepted the association. Otherwise (or upon any errors), it should return `NOTOK`.

When an activity associated with an association occurs, the event-handler routine is invoked with one parameter:

```
(*workfnx) (fd);
int     fd;
```

The parameter is:

`fd`: the association-descriptor of the association.

The event-handler should then call the appropriate routine to read the next event from the association (e.g., `RoWaitRequest` if remote operations are being used). The event-handler should return `NOTOK` if the association is terminated or aborted, or `OK` otherwise.

The `isodeserver` routine uses the `TNetAccept` routine to wait for the next event on existing associations and for new connections. It is possible, though unlikely for a failure to occur during this operation. In this event, the event-handler routine is invoked with one parameter:

```
(*losefnx) (td);
struct TSAPdisconnect *td;
```

The parameter is:

`td`: a pointer to an `TSAPdisconnect` structure updated by `TNetAccept`.

The event-handler should handle the error however it deems proper (usually by logging it and then returning).

An example of this facility is presented in Section 2.9 below.

Another interface to this approach is provided for servers that may be engaged in other operations besides answering incoming calls (e.g., initiating outgoing calls too). This interface provides indentical functionality to the `isodeserver` interface, but allows access to other events.

The interface is provided by two calls. The first is `iserver_init`.

```
int     iserver_init (argc, argv, aei, initfnx, td)
int     argc;
char  **argv;
AEI     aei;
IFP     initfnx;
struct TSAPdisconnect *td;
```

The parameters to this procedure are similar in part to the `isodeserver` procedure:

`argc`: the length of the argument vector which the program was invoked with;

`argv`: the argument vector which the program was invoked with;

`aei`: the information on the application-entity offering the service;

`initfnx`: the address of an event-handler routine to be invoked when a new connection arrives, and

`td`: a pointer to an `TSAPdisconnect` structure, which is updated only if the call fails.

This procedure registers a listener for the address and may may call the `initfnx` if the server is running in dynamic mode. If the call fails the `td` parameter will indicate the reason for failure.

One this function has been called, the `iserver_wait` procedure should be called at regular intervals to handle incoming events, typically in a loop of some kind. It is called as follows:

```
int     iserver_wait (initfnx, workfnx, losefnx, nfds,
                nfds, rfds, wfds, efds, secs, td)
IFP     initfnx,
        workfnx,
        losefnx;
int     nfds;
fd_set *rfds,
       *wfds,
       *efds;
int     secs;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`initfnx`: the address of an event-handler routine to be invoked when a new connection arrives;

`workfnx`: the address of an event-handler routine to be invoked when activity occurs on an association;

`losefnx`: the address of an event-handler routine to be invoked if `TNetAccept` (described in Section 4.7) fails;

`nfds/rfds/wfds/efds`: additional association-descriptors/file descriptors to await for activity on;

`secs`: the maximum number of seconds to wait for activity (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

`td`: a pointer to an `TSAPdisconnect` structure, which is updated only if the call fails.

This routine calls the `TNetAccept` routine to wait for incoming calls. It will call the procedures `initfnx`, `workfnx` and `losefnx` in an indetical way to the `isodeserver`. The routine returns on a number of conditions.

- If one of the supplied association/file descriptors registers activity the procedure returns with the mask updated.

- If an incoming association occurs. It should be accepted or rejected by the `initfnx` then the procedure will return.

- If some activity happens on one of the accepted calls. This is handled by the `workfnx` then the procedure will return.

- If the time given runs out, the procedure will return.

An outline usage of these procedures might be something like:

```
if (iserver_init (argc, argv, aei, ros_init, td) == NOTOK)
    /* error */
for (;;) {
    /* set up descriptors */
    switch (iserver_wait (ros_init, ros_work, ros_lose, nfds,
                &rfds, &wfds, NULLFD, timeout, td)) {
        case DONE:
            exit (0);

        case NOTOK:
            /* error */

        case OK:
            /* check fds for activity, do other things */
}
```

Please note that `isodeserver` and `iserver_wait` implement one possible discipline for association management. Many others are possible, depending on the needs of the service being provided. Further, note that while the text above and the example below are expressed in terms of association control (i.e., they make use of `AcInit` and `AcAssocResponse`), this facility will provide similar support at any other layer in the system (e.g., `isodeserver` can be used for transport or session entities).

Also note that as these routines use the `TNetAccept` routines, child process are collected automatically. If you wish to start child processes and wait for their exit status, you should take note of the warnings associated with the `TNetAccept` procedure (see Section 4.7 on page 130 of *Volume Two*).

2.6 Restrictions on User Data

To quote the [ISO88a] specification:

> **NOTE:** Use of the services ... may be subject to some constraints from (the) session services until work on providing unlimited length user data field parameters on session primitives is completed.

2.7 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `aci` parameter given to the routine. The `aci_abort` element of the `AcSAPindication` structure encodes the reason for the failure. One coerces the pointer to an `AcSAPabort` structure, and consults the `aca_reason` element of this latter structure. This element can be given as a parameter to the routine `AcErrString` which returns a null-terminated diagnostic string.

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

The macro can be used to determine if the failure is fatal (the association has been lost).

```
int     ACS_FATAL(r)
int     r;
```

For protocol purists, the `ACS_OFFICIAL` macro can be used to determine if the error is an "official" error as defined by the specification, or an "unofficial" error used by the implementation.

```
int     ACS_OFFICIAL (r)
int     r;
```

2.8 Compiling and Loading

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

2.9 An Example

One example of the use of the *libacsap*(3n) library is found in Section 3.6.1 on page 75. This example is straight-forward in presenting the interaction of association control and remote operations. Now let's consider how to rewrite the server to use the facilities described above in Section 2.5. Instead of using an asynchronous interface, a synchronous interface will be employed. Only Figure 3.1 from the example in Section 3.6.1 need be changed.

We assume that there are three exception-logging routines: *fatal*, which prints a diagnostic and terminates the process; *error*, which prints a diagnostic and then executes the statement

```
longjmp (toplevel, NOTOK);
```

and, *warn*, which simply prints a diagnostic.

In Figure 2.3, the replacement routines are shown. First, the application-entity title for the service is deteremined. The routine `isodeserver` is then called with its requisite arguments. If the routine fails, the process terminates after printing a diagnostic. Otherwise the process exits. In the case where the *tsapd*(8c) daemon invokes a new instance of the server each time an incoming connection is received, `isodeserver` will return after that assocation has been released. In the case where all incoming connections are given to a single instance of the server, then `isodeserver` returns only if a fatal error is detected.

The routine `ros_init` is called for each incoming connection. First, the ACSE state is re-captured by calling `AcInit`. If this succeeds, then any command line arguments are parsed. These arguments are present only if the server was invoked by the *tsapd*(8c) daemon. Assuming that the responder is satisfied with the proposed association, the routine then calls `AcAssocResponse` to accept the association. Then, `RoSetService` is called to set the underlying service to be used for remote operations. Finally, the association-descriptor is returned to `isodeserver`.

The routine `ros_work` is called when activity occurs on an association. The routine sets a global return vector using *setjmp* (3) and then calls `RoWaitRequest` to read the next indication. This usually results in the routine `ros_indication` (found in Figure 3.2 on page 77) being called. If the association was not released, then `ros_work` returns `OK`. Otherwise if some error occurred, use of the routine `error` will cause control to return to the `setjmp` call. In this case, `AcUAbortRequest` is called to make sure that the association is (ungracefully) released, then `NOTOK` is returned to `isodeserver`.

The routine `ros_lose` simply logs the failure of `TNetAccept` when called from `isodeserver`.

```
#include <stdio.h>
#include "generic.h"          /* defines OPERATIONS and ERRORS */
#include <isode/rosap.h>
#include <isode/tsap.h>       /* for listening */
#include <setjmp.h>


static char *myservice = "service";
static char *mycontext = "context";

static jmp_buf toplevel;


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


main (argc, argv, envp)
int     argc;
char  **argv,
      **envp;
{
    AEI     aei;
    struct TSAPdisconnect tds;
    register struct TSAPdisconnect *td = &tds;

    if ((aei = _str2aei (PLocalHostName (), myservice, mycontext, 0,
                         NULLCP, NULLCP)) == NULLAEI)
        fatal ("unable to resolve service: %s", PY_pepy);

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

    exit (0);
}

...
```

Figure 2.3: The generic ROS server

```
...

static int ros_init (vecp, vec)
int      vecp;
char  **vec;
{
    int      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) {
        warn ("initialization fails:  %s", AcErrString (aca -> aca_reason));
        return NOTOK;
    }

    sd = acs -> acs_sd;
    ACSFREE (acs);

/* read command line arguments here... */

    if (AcAssocResponse (sd, ACS_ACCEPT, ACS_USER_NULL, NULLOID, NULLAEI,
                    &ps -> ps_called, NULLPC, ps -> ps_defctxresult,
                    ps -> ps_prequirements, ps -> ps_srequirements,
                    SERIAL_NONE, ps -> ps_settings, &ps -> ps_connect,
                    NULLPEP, 0, aci) == NOTOK) {
        warn ("A-ASSOCIATE.RESPONSE: %s", AcErrString (aca -> aca_reason));
        return NOTOK;
    }

    if (RoSetService (sd, RoPService, roi) == NOTOK)
        fatal ("RoSetService:  %s", RoErrString (rop -> rop_reason));

    return sd;
}

...
```

Figure 2.3: The generic ROS server (continued)

```
...

static int  ros_work (fd)
int         fd;
{
    int         result;
    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:
            (void) AcUAbortRequest (fd, NULLPEP, 0, &acis);
            return NOTOK;
    }

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

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

    return OK;
}

...
```

Figure 2.3: The generic ROS server (continued)

```
...

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

Figure 2.3: The generic ROS server (continued)

2.10 For Further Reading

The ISO specification for association control services is defined in [ISO88a]. The corresponding protocol definition is [ISO88b].

Chapter 3

Remote Operations

The *librosap*(3n) library implements the Remote Operations Service (ROS). Three service disciplines are implemented: when the ECMA interpretation of the ROS is used, we term this the *basic* service discipline; when the CCITT X.400 interpretation is used, we term this the *advanced* service discipline; and, when the new ISO and CCITT MOTIS interpretation is used, we term this the *complete* service discipline. The advanced discipline is somewhat less restrictive than the basic discipline, at the cost of requiring a more complex implementation on the part of both the provider and the user. The complete discipline, in addition to all of the facilities provided by the advanced discipline, also supports the notion of *linked* operations.

The abstraction provided to applications is that of an *association* for remote operations. An association is a binding between two users: the *initiator* of the association, and the *responder* to the association. Once an association is established, the initiator requests the responder to perform remote operations. The responder in turn attempts these operations, replying with either a result or an error. This process continues until the initiator decides to release the association.

Like most models of OSI services, the underlying assumption is one of an asynchronous environment: the service provider may generate events for the service user without the latter entity triggering the actions which led to the event. For example, in a synchronous environment, an indication that data has arrived usually occurs only when the service user asks the service provider to read data; in an asynchronous environment, the service provider may interrupt the service user at any time to announce the arrival of data.

The `rosap` module in this release presents a synchronous interface. However once the association is established, an asynchronous interface may be selected.

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

3.1 Notice

Please read the following important message.

NOTE:	The interface described herein is a "raw" interface to the remote operations service. Consult *Volume Four* for a "cooked" interface.

3.2 Service Disciplines and Associations

There are three service disciplines for remote operations: *basic*, *remote*, and *complete*. The basic service discipline limits its users by permitting only the initiator to invoke remote operations. Certain applications, e.g., message handling systems, require more freedom than this, along with more reliability.

The Remote Operations Service Element does not establish associations. Consult Section 2.1 to determine which mechanism you should use to manage associations for your application. Since the advanced and complete disciplines are both proper supersets of the basic service discipline, two users of an association can utilize exactly the features of the basic service discipline even though the advanced or complete service discipline has been selected, without a loss of generality.

3.3 Remote Operations

Once the association has been established, an association-descriptor is used to reference the association. This is usually the first parameter given to any of the remaining routines in the *librosap*(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.

```
struct RoSAPindication {
    int     roi_type;
#define ROI_INVOKE      0x00
#define ROI_RESULT      0x01
#define ROI_ERROR       0x02
#define ROI_UREJECT     0x03
#define ROI_PREJECT     0x04
#define ROI_END         0x05
#define ROI_FINISH      0x06

    union {
        struct RoSAPinvoke roi_un_invoke;
        struct RoSAPresult roi_un_result;
        struct RoSAPerror roi_un_error;
        struct RoSAPureject roi_un_ureject;
        struct RoSAPpreject roi_un_preject;
        struct RoSAPend roi_un_end;
        struct AcSAPfinish roi_un_finish;
    }   roi_un;
#define roi_invoke      roi_un.roi_un_invoke
#define roi_result      roi_un.roi_un_result
#define roi_error       roi_un.roi_un_error
#define roi_ureject     roi_un.roi_un_ureject
#define roi_preject     roi_un.roi_un_preject
#define roi_end         roi_un.roi_un_end
#define roi_finish      roi_un.roi_un_finish
};
```

As shown, this structure is really a discriminated union (a structure with a tag element followed by a union). Hence, on a failure return, one first coerces a pointer to the `RoSAPpreject` structure contained therein, and then consults the elements of that structure.

```
struct RoSAPpreject {
    int     rop_reason;

    int     rop_id;
    PE      rop_apdu;

#define ROP_SIZE        512
```

```
        int     rop_cc;
        char    rop_data[ROP_SIZE];
};
```

The elements of a RoSAPpreject structure are:

- `rop_reason`: the reason for the provider-initiated reject (codes are listed in Table 3.1),
- `rop_id/rop_apdu`: if an APDU could not be transferred (`rop_reason` is `ROS_APDU`), then this is the invocation ID of the APDU and (possibly) the APDU itself, respectively; and,
- `rop_data/rop_cc`: a diagnostic string from the provider.

Note that the `rop_apdu` element is allocated via *malloc*(3) and should be released using the `ROPFREE` macro when no longer referenced. The `ROPFREE` macro behaves as if it was defined as:

```
void    ROPFREE (rop)
struct RoSAPpreject *rop;
```

The macro frees only the data allocated in the `RoSAPpreject` structure and not the structure itself.

On a failure return, if the `rop_reason` element of the `RoSAPpreject` structure is associated with a fatal error, the the association is released. The `ROS_FATAL` macro can be used to determine this.

```
int     ROS_FATAL (r)
int     r;
```

For protocol purists, the `ROS_OFFICIAL` macro can be used to determine if the error is an "official" error as defined by the specification, or an "unofficial" error used by the implementation.

```
int     ROS_OFFICIAL (r)
int     r;
```

Finally, some of these routines also take a `priority` parameter indicating the relative importance of the remote operation. Under the basic service discipline, this parameter is ignored. Under the advanced or complete service discipline, each application decides on its own integer-valued definitions. The manifest constant `ROS_NOPRIO` can be used if the application is unconcerned with the priority.

Provider-Initiated Aborts (fatal)	
ROS_ADDRESS	Address unknown
ROS_REFUSED	Connect request refused on this network connection
ROS_SESSION	Session disconnect
ROS_PROTOCOL	Protocol error
ROS_CONGEST	Congestion at RoSAP
ROS_REMOTE	Remote system problem
ROS_DONE	Assocation done via async handler
ROS_ABORTED	Peer aborted association
ROS_RTS	RTS disconnect
ROS_PRESENTATION	Presentation disconnect
ROS_ACS	ACS disconnect
Provider-Initiated Rejects (non-fatal)	
ROS_GP_UNRECOG	Unrecognized APDU
ROS_GP_MISTYPED	Mistyped APDU
ROS_GP_STRUCT	Badly structured APDU
User-Initiated Rejects (fatal)	
ROS_VALIDATE	Authentication failure
ROS_BUSY	Busy

Table 3.1: RoSAP Failure Codes

User-Initiated Rejects (non-fatal)	
`ROS_IP_DUP`	Duplicate invocation
`ROS_IP_UNRECOG`	Unrecognized operation
`ROS_IP_MISTYPED`	Mistyped argument
`ROS_IP_LIMIT`	Resource limitation
`ROS_IP_RELEASE`	Initiator releasing
`ROS_IP_UNLINKED`	Unrecognized linked ID
`ROS_IP_LINKED`	Linked response unexpected
`ROS_IP_CHILD`	Unexpected child operation
`ROS_RRP_UNRECOG`	Unrecognized invocation
`ROS_RRP_UNEXP`	Result response unexpected
`ROS_RRP_MISTYPED`	Mistyped result
`ROS_REP_UNRECOG`	Unrecognized invocation
`ROS_REP_UNEXP`	Error response unexpected
`ROS_REP_RECERR`	Unrecognized error
`ROS_REP_UNEXPERR`	Unexpected error
`ROS_REP_MISTYPED`	Mistyped parameter
Interface Errors (non-fatal)	
`ROS_PARAMETER`	Invalid parameter
`ROS_OPERATION`	Invalid operation
`ROS_TIMER`	Timer expired
`ROS_WAITING`	Indications waiting
`ROS_APDU`	APDU not transferred
`ROS_INTERRUPTED`	Stub interrupted

Table 3.1: RoSAP Failure Codes (continued)

3.3.1 Selecting an Underlying Service

As should be obvious from Section 3.2, the *librosap*(3n) library supports the use of three different underlying services, depending on the method used to establish the association. The *librosap*(3n) library is constructed in such a way as to avoid loading the object code for all three underlying services when only one is desired. In order to effect this, it is necessary to give the loader a small hint.

Once an association has been established, the `RoSetService` routine should be called.

```
int     RoSetService (sd, bfunc, roi)
int     sd;
IFP     bfunc;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`bfunc`: a *magic* argument, use:

Value	**Underlying Service**	**Routine**

`RoRtService`	Reliable Transfer	`RtOpenResponse`
		`RtOpenRequest`
		`RtBeginResponse`
		`RtBeginRequest`

`RoPService`	Presentation	`AcAssocResponse` `AcAssocRequest`
`RoSService`	Session	`RoBeginResponse` `RoBeginRequest`

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

The call to `RoSetService` should be made after a successful return from any of the routines listed above.

3.3.2 Invoking Operations

The `RoInvokeRequest` routine is used to request an operation to be performed remotely, and corresponds to a RO-INVOKE.REQUEST action. Under the basic service discipline, this action may be taken by only the initiator of an association; under the advanced or complete service discipline, no such restriction is made.

```
int     RoInvokeRequest (sd, op, class, args, invoke,
                linked, priority, roi)
int     sd;
int     op,
        class,
        invoke,
       *linked,
        priority;
PE      args;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`op`: the operation code;

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

`args`: the arguments for the operation;

`invoke`: the invocation ID of this request;

`linked`: the linked ID of this request (only present if the complete service discipline has been selected, use `NULLIP` otherwise);

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

`roi`: a pointer to a `RoSAPindication` structure, which is always updated on synchronous operations, and only updated if the call fails for asynchronous operations.

If the class parameter was ROS_ASYNC, then RoInvokeRequest returns immediately. Otherwise, after queuing the request, the RoWaitRequest routine (described in Section 3.3.4 on page 66) is called implicitly to return the reply of the request. Every attempt will be made to return the corresponding reply to the request. Nevertheless, it is the responsibility of the user to verify the invocation ID which is returned.

The routine RoIntrRequest is similar to RoInvokeRequest with a class argument of ROS_SYNC: it invokes an operation and then waits for a response. However, if the user generates an interrupt, usually by typing control-C ('^C'), then RoIntrRequest will return immediately by simulating a RO-U-REJECT.INDICATION with reason ROS_INTERRPUTED (see section 3.3.4 on page 70). This is useful for users of remote operations which support an "abandon" functionality (e.g., the OSI Directory).

```
int     RoIntrRequest (sd, op, args, invoke, linked,
                priority, roi)
int     sd;
int     op,
        invoke,
       *linked,
        priority;
PE      args;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

sd: the association-descriptor;

op: the operation code;

args: the arguments for the operation;

invoke: the invocation ID of this request;

linked: the linked ID of this request (only present if the complete service discipline has been selected, use NULLIP otherwise);

priority: the priority of this request (use ROS_NOPRIO if the priority is undetermined); and, `roi`: a pointer to a `RoSAPindication` structure, which is always updated on synchronous operations, and only updated if the call fails for asynchronous operations.

3.3.3 Replying to Requests

When a request to perform a remote operation has been received by the responder to an association, the responder either returns a result or an error (or in some cases, returns nothing at all).

The `RoResultRequest` routine is used to return a result, and corresponds to the RO-RESULT.REQUEST action. Under the basic service discipline, this action may only be taken by the responder to an association; under the advanced or complete service discipline, no such restriction is made.

```
int     RoResultRequest (sd, invoke, op, result, priority, roi)
int     sd;
int     invoke,
        op,
        priority;
PE      result;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`invoke`: the invocation ID of the request corresponding to this reply;

`op`: the operation code of the operation performed (meaningful only in the complete service discipline);

`result`: the results of the operation;

`priority`: the priority of this reply; and,

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

The `RoErrorRequest` routine is used to return an error, and corresponds to the RO-ERROR.REQUEST action. Under the basic service discipline, this action may only be taken by the responder to an association; under the advanced or complete service discipline, no such restriction is made.

```
int     RoErrorRequest (sd, invoke, error, params,
                priority, roi)
int     sd;
int     invoke,
        error,
        priority;
PE      params;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`invoke`: the invocation ID of the request corresponding to this reply;

`op`: the error code;

`params`: the parameters for the error;

`priority`: the priority of this request; and,

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

3.3.4 Reading Replies

The `RoWaitRequest` routine is used to await either a request or a reply from the other user.

```
int     RoWaitRequest (sd, secs, roi)
int     sd;
int     secs;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`secs`: the maximum number of seconds to wait for the data (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 always updated.

Unlike the other routines in the *librosap*(3n) library, the RoWaitRequest routine returns one of three values: NOTOK (on failure), OK (on reading a request or a reply), or DONE (when something else happens).

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

Otherwise if the call to RoWaitRequest returns the manifest constant OK, then a request or a reply has arrived. This event is encoded in the roi parameter, depending on the value of the roi_type element. Currently, when RoWaitRequest returns OK, the roi_type element is set to one of four values:

Value	Event
ROI_INVOKE	RO-INVOKE.INDICATION
ROI_RESULT	RO-RESULT.INDICATION
ROI_ERROR	RO-ERROR.INDICATION
ROI_UREJECT	RO-U-REJECT.INDICATION

Otherwise if the call to RoWaitRequest returns the manifest constant DONE, then some event other than a request or reply arriving has occurred. This event is encoded in the roi parameter, depending on the value of the roi_type element. Currently, when RoWaitRequest returns DONE, the roi_type element is set to one of two values:

Value	Event
ROI_END	RO-END.INDICATION (for old-style associations)
ROI_FINISH	A-RELEASE.INDICATION (or RT-CLOSE.INDICATION)

Invocation Indication

When an RO-INVOKE.INDICATION event occurs, the roi_type field of the roi parameter contains the value ROI_INVOKE, and a RoSAPinvoke structure is contained inside the roi parameter.

```
struct RoSAPinvoke {
    int     rox_id;

    int     rox_linkid;
    int     rox_nolinked;

    int     rox_op;
    PE      rox_args;
};
```

The elements of this structure are:

rox_id: the invocation ID of this request;

rox_linkid: if rox_nolinked is not set, then this contains the invocation ID of the linked request;

rox_nolinked: the linked ID indicator (if set, then this invocation is not linked to another operation);

rox_op: the operation code; and,

rox_args: the arguments for the operation.

Note that the rox_data element is allocated via *malloc*(3) and should be released using the ROXFREE macro when no longer referenced. The ROXFREE macro behaves as if it was defined as:

```
void    ROXFREE (rox)
struct RoSAPinvoke *rox;
```

The macro frees only the data allocated in the RoSAPinvoke structure and not the structure itself.

Under the basic service discipline, only the responder to an association will receive this event; it is expected that the user will (eventually) call either the RoResultRequest, the RoErrorRequest, or perhaps the RoURejectRequest routine in response.

Result Indication

When an RO-RESULT.INDICATION event occurs, the `roi_type` field of the `roi` parameter contains the value `ROI_RESULT`, and a `RoSAPresult` structure is contained inside the `roi` parameter.

```
struct RoSAPresult {
    int     ror_id;

    PE      ror_result;
};
```

The elements of this structure are:

`ror_id`: the invocation ID of this reply, which is identical to the ID of the request which generated the results;

`ror_op`: the operation code of the operation performed (meaningful only in the complete service discipline); and,

`ror_result`: the results of the operation.

Note that the `ror_result` element is allocated via *malloc*(3) and should be released using the `RORFREE` macro when no longer referenced. The `RORFREE` macro behaves as if it was defined as:

```
void    RORFREE (ror)
struct RoSAPresult *ror;
```

The macro frees only the data allocated in the `RoSAPresult` structure and not the structure itself.

Under the basic service discipline, only the initiator to an association will receive this event in response to an earlier call to `RoInvokeRequest`.

Error Indication

When an RO-ERROR.INDICATION event occurs, the `roi_type` field of the `roi` parameter contains the value `ROI_ERROR`, and a `RoSAPerror` structure is contained inside the `roi` parameter.

```
struct RoSAPerror {
    int     roe_id;

    int     roe_error;
    PE      roe_param;
};
```

The elements of this structure are:

`roe_id`: the invocation ID of this reply, which is identical to the ID of the request which generated the error;

`roe_error`: the error code; and,

`roe_param`: the parameters of the error.

Note that the `roe_param` element is allocated via *malloc*(3) and should be released using the `ROEFREE` macro when no longer referenced. The `ROEFREE` macro behaves as if it was defined as:

```
void    ROEFREE (roe)
struct RoSAPerror *roe;
```

The macro frees only the data allocated in the `RoSAPerror` structure and not the structure itself.

Under the basic service discipline, only the initiator to an association will receive this event in response to an earlier call to `RoInvokeRequest`.

User-Reject Indication

When an RO-U-REJECT.INDICATION event occurs, the `roi_type` field of the `roi` parameter contains the value `ROI_UREJECT`, and a `RoSAPureject` structure is contained inside the `roi` parameter.

```
struct RoSAPureject {
    int     rou_id;
    int     rou_noid;

    int     rou_reason;
};
```

The elements of this structure are:

`rou_id`: if `rou_noid` is not set, then this contains the invocation ID of the request which generated the rejection;

`rou_noid`: the invocation ID indicator (if set, then no request in particular caused the rejection to be generated); and,

`rou_reason`: the reason for the rejection (a "non-fatal" user-initiated rejection code listed in Table 3.1).

End Indication

When the `roi_type` field of the `roi` parameter contains the value `ROI_END`, a `RoSAPend` structure is contained inside the `roi` parameter.

```
struct RoSAPend {
    int     roe_dummy;
};
```

Depending on whether the reliable transfer service was used to start the association, a X.400 CLOSE.INDICATION or a RO-END.INDICATION event has occurred, and the user should respond appropriately.

Finish Indication

When the `roi_type` field of the `RoSAPindication` parameter `roi` contains the value `ROI_FINISH`, a `AcSAPfinish` structure is contained inside the `roi` parameter. Depending on whether the reliable transfer service was used to start the association, an RT-CLOSE.INDICATION or an A-RELEASE.INDICATION event has occurred, and the user should respond appropriately.

3.3.5 Rejecting Requests and Replies

The `RoURejectRequest` routine is used to perform user-level error-recovery. Usually, it signals the rejection of a previously received request or reply.

```
int     RoURejectRequest (sd, invoke, reason, priority, roi)
int     sd;
int    *invoke,
```

```
            reason,
            priority;
    struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`invoke`: a pointer to the invocation ID of the request in question (or `NULLIP` if this rejection does not apply to a particular request or reply);

`reason`: a "non-fatal" user-initiated rejection code as listed in Table 3.1 on page 60;

`priority`: the priority of this request; and,

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

Upon a successful return from this call, a ROU-U-REJECT.INDICATION event has been queued for the other user.

3.3.6 Asynchronous Event Handling

The request/reply events discussed thus far have been synchronous in nature. Some users of the remote operations service may wish an asynchronous interface. The `RoSetIndications` routine is used to change the service associated with an association-descriptor to an asynchronous interface.

```
    int   RoSetIndications (sd, indication, roi)
    int   sd;
    IFP   indication;
    struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`indication`: the address of an event-handler routine to be invoked when an event occurs; and, `roi`: a pointer to a `RoSAPindication` structure, which is updated only if the call fails.

If the service is to be made asynchronous, then `indication` is specified; otherwise, if the service is to be made synchronous, it is not specified (use the manifest constant `NULLIFP`). The most likely reason for the call failing is `ROS_WAITING`, which indicates that an event is waiting for the user.

When an event occurs, the event-handler routine is invoked with two parameters:

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

The parameters are:

`sd`: the association-descriptor; and,

`roi`: a pointer to a `RoSAPindication` structure encoding the event.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released with the appropriate macro (i.e., `ROPFREE`, `ROXFREE`, `RORFREE`, or `ROEFREE`) when no longer needed.

When an event-handler is invoked, future invocations of the event-hander are blocked until it returns. The return value of the event-handler is ignored. Further, during the execution of a synchronous call to the library, the event-handler will be blocked from being invoked. The one exception to this is a call to `RoInvokeRequest` with the `class` parameter set to `ROS_SYNC`. In this circumstance, replies to invocations other than the one being waited for will result in the event-handler being invoked as appropriate.

NOTE:	The *librosap*(3n) library uses the SIGEMT signal to provide these services. Programs using asynchronous association-descriptors should NOT use SIGEMT for other purposes.

3.3.7 Synchronous Event Multiplexing

A user of the remote operations service may wish to manage multiple association-descriptors simultaneously; the routine `RoSelectMask` is provided for this purpose. This routine updates a file-descriptor mask and associated counter for use with `xselect` (consult Section 2.4), as assocation-descriptors are file-descriptors.

```
int     RoSelectMask (sd, mask, nfds, roi)
int     sd;
fd_set *mask;
int    *nfds;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`mask`: a pointer to a file-descriptor mask meaningful to `xselect`;

`nfds`: a pointer to an integer-valued location meaningful to `xselect`; and,

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

If the call is successful, then the `mask` and `nfds` parameters can be used as arguments to `xselect`. The most likely reason for the call failing is `ROS_WAITING`, which indicates that an event is waiting for the user.

If `xselect` indicates that the association-descriptor is ready for reading, `RoWaitRequest` should be called with the `secs` parameter equal to `OK`. If the network activity does not constitute an entire event for the user, then `RoWaitRequest` will return `NOTOK` with error code `ROS_TIMER`.

3.4 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;
```

3.5 Compiling and Loading

Programs using the *librosap*(3n) library should include `<isode/rosap.h>`, which automatically includes the header file `<isode/psap.h>` described in Chapter 5 and the header file `<isode/acsap.h>` described in Chapter 2. The programs should also be loaded with `-lisode`.

3.6 Two Examples

Two examples are now presented: a detailed exposition on the construction of a responder for remote operations; and, a terse presentation of an initiator process.

3.6.1 The Generic Server

Let's consider how one might construct a generic server which uses remote operations services. This entity will use an asynchronous interface.

There are two parts to the program: initialization and the request/reply loop. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure 3.1, the initialization steps for the generic responder, including the outer *C* wrapper, is shown. First, the ACSE state is re-captured by calling `AcInit`. If this succeeds, then any command line arguments are parsed. Assuming that the responder is satisfied with the proposed association, it calls `AcAssocResponse` to accept the association. Then, `RoSetService` is called to set the underlying service to be used for remote operations. The `RoSetIndications` routine is called to specify an asynchronous event handler. Finally, the main request/reply loop is realized. The server simply uses *pause*(2) to wait for the next event.

In Figure 3.2 on page 77, the `ros_indication` routine simply dispatches based on the value of the `roi_type` element of the `RoSAPindication` structure.

```
#include <stdio.h>
#include "generic.h"              /* defines OPERATIONS and ERRORS */
#include <isode/rosap.h>


int     ros_indication ();


main (argc, argv, envp)
int     argc;
char  **argv,
      **envp;
{
    int     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 (argc, argv, acs, aci) == NOTOK)
        error ("initialization fails:  %s", AcErrString (aca -> aca_reason));

    sd = acs -> acs_sd;
    ACSFREE (acs);

/* read command line arguments here... */

    if (AcAssocResponse (sd, ACS_ACCEPT, ACS_USER_NULL, NULLOID, NULLAEI,
                &ps -> ps_called, NULLPC, ps -> ps_defctxresult,
                ps -> ps_prequirements, ps -> ps_srequirements,
                SERIAL_NONE, ps -> ps_settings, &ps -> ps_connect,
                NULLPEP, 0, aci) == NOTOK)
        error ("A-ASSOCIATE.RESPONSE: %s", AcErrString (aca -> aca_reason));

    if (RoSetService (sd, RoPService, roi) == NOTOK)
        error ("RoSetService:  %s", RoErrString (rop -> rop_reason));

    if (RoSetIndications (sd, ros_indication, roi) == NOTOK)
        error ("RoSetIndications:  %s", RoErrString (rop -> rop_reason));

    for (;;)
        pause ();
}
```

Figure 3.1: Initializing the generic ROS responder

```
...

static int ros_indication (sd, roi)
int     sd;
register struct RoSAPindication *roi;
{
   switch (roi -> roi_type) {
         case ROI_INVOKE:
             ros_invoke (sd, &roi -> roi_invoke);
             break;

         case ROI_RESULT:
             ros_result (sd, &roi -> roi_result);
             break;

         case ROI_ERROR:
             ros_error (sd, &roi -> roi_error);
             break;

         case ROI_UREJECT:
             ros_ureject (sd, &roi -> roi_ureject);
             break;

         case ROI_PREJECT:
             ros_preject (sd, &roi -> roi_preject);
             break;

         case ROI_FINISH:
             ros_finish (sd, &roi -> roi_finish);
             break;

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

...
```

Figure 3.2: The request/reply loop for the generic ROS responder

```
...

int     OP1 (), ..., OPn ();


/* OPERATIONS are numbered APDU_OPx, where each is a unique integer.  Further,
   APDU_UNKNOWN is used as a tag different than any valid operation.

   ERRORS are numbered ERROR_xyz, where each is a unique integer.
   ERROR_MISTYPED is used to signal an argument error to an operation.
   Further, ERROR_UNKNOWN is used as a tag to indicate that the operation
   succeeded.

   Finally, note that rox -> rox_args is updated in place by these routines.
   If the routine returns ERROR_UNKNOWN, then rox_args contains the results
   of the operation.  If the routine returns ERROR_MISTYPED, then rox_args is
   untouched.  Otherwise, if the routine returns any other value, then
   rox_args contains the parameters of the error which occurred.  Obviously,
   each routine calls ROXFREE prior to setting rox_args to a new value.
 */

static struct dispatch {
    int     ds_operation;
    IFP     ds_vector;
}       dispatches[] = {
    APDU_OP1,   OP1,

...

    APDU_OPn,   OPn,

    APDU_UNKNOWN
};

...
```

Figure 3.2: The request/reply loop for the generic ROS responder (continued)

```
...

static int ros_invoke (sd, rox)
int     sd;
register struct RoSAPinvoke *rox;
{
   int     result;
   register struct dispatch   *ds;
   struct RoSAPindication  rois;
   register struct RoSAPindication *roi = &rois;
   register struct RoSAPpreject   *rop = &roi -> roi_preject;

   for (ds = dispatches; ds -> ds_operation != APDU_UNKNOWN; ds++)
         if (ds -> ds_operation == rox -> rox_op)
             break;

   if (ds -> ds_operation == APDU_UNKNOWN) {
         if (RoURejectRequest (sd, &rox -> rox_id, ROS_IP_UNRECOG,
                     ROS_NOPRIO, roi) == NOTOK)
             error ("RO-U-REJECT.REQUEST: %s", RoErrString (rop -> rop_reason));
         goto out;
   }

   if (rox -> rox_nolinked == 0) {
         if (RoURejectRequest (sd, &rox -> rox_id, ROS_IP_LINKED,
                     ROS_NOPRIO, roi) == NOTOK)
             error ("RO-U-REJECT.REQUEST: %s", RoErrString (rop -> rop_reason));
         goto out;
   }

   switch (result = (*ds -> ds_vector) (rox)) {
         case ERROR_UNKNOWN:
             if (RoResultRequest (sd, rox -> rox_id, rox -> rox_op,
                         rox -> rox_args, ROS_NOPRIO, roi) == NOTOK)
                   error ("RO-RESULT.REQUEST: %s",
                         RoErrString (rop -> rop_reason));
             break;

         default:
             if (RoErrorRequest (sd, rox -> rox_id, result, rox -> rox_args,
                         ROS_NOPRIO, roi) == NOTOK)
                   error ("RO-ERROR.REQUEST: %s",
                         RoErrString (rop -> rop_reason));
             break;

         case ERROR_MISTYPED:
             if (RoURejectRequest (sd, &rox -> rox_id, ROS_IP_MISTYPED,
                         ROS_NOPRIO, roi) == NOTOK)
                   error ("RO-U-REJECT.REQUEST: %s",
                         RoErrString (rop -> rop_reason));
             break;
   }

out: ;
   ROXFREE (rox);
}

...
```

Figure 3.2: The request/reply loop for the generic ROS responder (continued)

```
...

static int ros_result (sd, ror)
int     sd;
register struct RoSAPresult *ror;
{
    struct RoSAPindication rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (RoURejectRequest (sd, &ror -> ror_id, ROS_RRP_UNRECOG, ROS_NOPRIO, roi)
            == NOTOK)
        error ("RO-U-REJECT.REQUEST: %s", RoErrString (rop -> rop_reason));

    RORFREE (ror);
}


static int ros_error (sd, roe)
int     sd;
register struct RoSAPerror *roe;
{
    struct RoSAPindication rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (RoURejectRequest (sd, &roe -> roe_id, ROS_REP_UNRECOG, ROS_NOPRIO, roi)
            == NOTOK)
        error ("RO-U-REJECT.REQUEST: %s", RoErrString (rop -> rop_reason));

    ROEFREE (roe);
}


static int ros_ureject (sd, rou)
int     sd;
register struct RoSAPureject *rou;
{
/* handle rejection here... */
}


static int ros_preject (sd, rop)
int     sd;
register struct RoSAPpreject *rop;
{
    if (ROS_FATAL (rop -> rop_reason))
        error ("RO-REJECT-P.INDICATION: %s", RoErrString (rop -> rop_reason));

/* handle temporary failure here... */
}

...
```

Figure 3.2: The request/reply loop for the generic ROS responder (continued)

If the event was caused by a request to invoke an operation, then `ros_invoke` is called. This routine consults a dispatch table to find the function which will execute the operation. Based on the return value of the function, either the result is returned, an error is returned, or the request to perform the operation is rejected.

If the event was caused by a reply to the invocation of an operation (i.e., either results or an error), this is rejected. The basic service discipline prohibits this event from happening to responders; hence, the `ros_result` and `ros_error` routines are used as stubs.

If the event was caused by a rejection of a previous reply, then the `ros_ureject` routine is called to handle this. If the event was caused by the provider initiating a rejection, then the `ros_preject` routine is called to handle this.

Finally, if the event was caused by the association being released, the `ros_finish` routine is called to handle this.

3.6.2 The Generic Client

As the previous example described — in great detail — how users of the remote operations services employ the *librosap*(3n) library, we now present a short example of how a client connects to the server.

In Figure 3.3, the initialization steps for the generic initiator, including the outer *C* wrapper, is shown. First, the application-entity information and presentation address are looked-up. Second, the application context and protocol control information objects are looked-up (and copied, since `ode2oid` returns a pointer to a static area). Next, the session connection reference is initialized. Then, the call to `AcAssocRequest` is made. The arguments to this routine are the minimal required for remote operations. If the call succeeds, then the client checks to see if the association is successfully established. If so, the association descriptor is captured, and then the routine `invoke`, which is not described in this example, is called. This routine presumably requests remote operations from the responder previously described. Upon the return of the `invoke` routine, the association is (forcibly) released, and the program exits.

```
...

static int  ros_finish (sd, acf)
int     sd;
struct AcSAPfinish *acf;
{
    struct AcSAPindication  acis;
    register struct AcSAPabort *aca = &acis.aci_abort;

    ACFFREE (acf);

    if (AcRelResponse (sd, ACS_ACCEPT, ACR_NORMAL, NULLPEP, 0, &acis) == NOTOK)
        error ("A-RELEASE.RESPONSE: %s", AcErrString (aca -> aca_reason));

    error ("association released");
}
```

Figure 3.2: The request/reply loop for the generic ROS responder (continued)

```
#include <stdio.h>
#include "generic.h"            /* defines OPERATIONS and ERRORS */
#include <isode/rosap.h>


static char *myservice = "service";

static char *mycontext = "iso service";
static char *mypci = "service pci version m.n";


main (argc, argv, envp)
int     argc;
char  **argv,
      **envp;
{
    int     sd;
    struct SSAPref sfs;
    register struct SSAPref *sf;
    register struct PSAPaddr *pa;
    struct AcSAPconnect accs;
    register struct AcSAPconnect   *acc = &accs;
    struct AcSAPrelease acrs;
    register struct AcSAPrelease   *acr = &acrs;
    struct AcSAPindication  acis;
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort *aca = &aci -> aci_abort;
    struct RoSAPindication  rois;
    register struct RoSAPpreject   *rop = &rois.roi_preject;
    register AEI aei;
    register OID ctx, pci;
    struct PSAPctxlist pcs;
    register struct PSAPctxlist *pc = &pcs;

    if ((aei = _str2aei (argv[1], myservice, mycontext, 0,
                                NULLCP, NULLCP)) == NULLAEI)
        error ("unable to resolve service: %s", PY_pepy);
    if ((pa = aei2addr (aei)) == NULLPA)
        error ("address translation failed");
    if ((ctx = ode2oid (mycontext)) == NULLOID)
        error ("%s: unknown object descriptor", mycontext);
    if ((ctx = oid_cpy (ctx)) == NULLOID)
        error ("oid_cpy");
    if ((pci = ode2oid (mypci)) == NULLOID)
        error ("%s: unknown object descriptor", mypci);
    if ((pci = oid_cpy (pci)) == NULLOID)
        error ("oid_cpy");
    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;

...
```

Figure 3.3: Initializing the generic ROS initiator

3.7 For Further Reading

The ECMA technical report on remote operation services is defined in [ECM85]. The CCITT recommendation describing remote operations, as supported by the reliable transfer service, is [CCI84b]. These both assume the use of old-style associations; the draft CCITT work which assumes the use of new-style associations is defined in [CCI88c] and [CCI88d]. Similarly, the corresponding ISO work is [ISO88e] and [ISO88f].

```
...

    if ((sf = addr2ref (PLocalHostName ())) == NULL) {
        sf = &sfs;
        (void) bzero ((char *) sf, sizeof *sf);
    }

/* read command line arguments here... */

    if (AcAssocRequest (ctx, NULLAEI, NULLAEI, NULLPA, pa, pc, NULLOID,
                0, ROS_MYREQUIRE, SERIAL_NONE, 0, sf, NULLPEP, 0, NULLQOS,
                acc, aci)
            == NOTOK)
        error ("A-ASSOCIATE.REQUEST: %s", AcErrString (aca -> aca_reason));

    if (acc -> acc_result != ACS_ACCEPT)
        error ("association rejected:  %s", AcErrString (aca -> aca_reason));

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

    if (RoSetService (sd, RoPService, &rois) == NOTOK)
        error ("RoSetService:  %s", RoErrString (rop -> rop_reason));

    invoke (sd);                    /* invoke the operations, etc. */

    if (AcRelRequest (sd, ACF_NORMAL, NULLPEP, 0, NOTOK, acr, aci) == NOTOK)
        error ("A-RELEASE.REQUEST: %s", AcErrString (aca -> aca_reason));

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

    ACRFREE (acr);

    exit (0);
}
```

Figure 3.3: Initializing the generic ROS initiator (continued)

Chapter 4

Reliable Transfer

The *librtsap*(3n) library implements the reliable transfer service (RTS). The abstraction provided to applications is that of an *association* for reliably transfering data. Most applications use the remote operations facilities, described in the previous chapter, and do not directly use the reliable transfer service.[1] However, for those applications which do not base themselves in remote operations, the reliable transfer interface is used.

As with most models of OSI services, the underlying assumption is one of a symmetric, asynchronous environment. That is, although peers exist at a given layer, one does not necessary view a peer as either a client or a server. Further, the service provider may generate events for the service user without the latter entity triggering the actions which led to the event. For example, in a synchronous environment, an indication that data has arrived usually occurs only when the service user asks the service provider to read data; in an asynchronous environment, the service provider may interrupt the service user at any time to announce the arrival of data.

The `rtsap` module in this release initially uses a client/server paradigm to start communications. Once the connection is established, a symmetric view is taken. In addition, initially the interface is synchronous; however once the connection is established, an asynchronous interface may be selected.

All of the routines in the *librtsap*(3n) library are integer-valued. They

[1] Actually, applications such as message-handling systems explicitly use the reliable transfer service to perform association management, and then optionally utilize the remote operations service for actual communication (otherwise they use reliable transfer directly). Both usages are permitted and encouraged in this implementation.

return the manifest constant `OK` on success, or `NOTOK` otherwise.

4.1 Associations

As briefly mentioned earlier, an association is the binding between two applications. An association is formed when one application, termed the *initiator*, specifies the address of a *responder* and asks the reliable transfer service to establish a connection.

There are three aspects of association management: *association establishment*, *association release*, and, *association abort*. Each of these are now described in turn. All of these facilities rely on the mechanisms described in Section 2.2 on page 17.

4.1.1 Association Establishment

The *librtsap*(3n) library distinguishes between the user which started an assocoation, the *initiator*, and the user which was subsequently bound to the association, the *responder*. We sometimes term these two entities the *client* and the *server*, respectively.

Addresses

Addresses for the reliable transfer service entity consist of two parts: a presentation address (as discussed in Section 2.2 on page 16 of *Volume Two*), and application-entity information (as discussed in Section 2.2.1 on page 17).

In Figure 2.1 on page 21, an example of how these components are determined is given.

Server Initialization

The *tsapd*(8c) daemon, upon accepting a connection from an initiating host, consults the ISO services database to determine which program on the local system implements the desired application context. Once the program has been ascertained, the daemon runs the program with any argument listed in the database. In addition, it appends some *magic arguments* to the argument vector. Hence, the very first action performed by the responder is to recapcture the RTSE state contained in the magic arguments. This is done by calling the routine RtInit, which on a successful return, is equivalent to a RT-OPEN.INDICATION from the reliable transfer service provider.

```
int     RtInit (vecp, vec, rts, rti)
int     vecp;
char  **vec;
struct RtSAPstart *rts;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

vecp: the length of the argument vector;

vec: the argument vector;

rts: a pointer to a RtSAPstart structure, which is updated only if the call succeeds; and,

rti: a pointer to a RtSAPindication structure, which is updated only if the call fails.

If RtInit is successful, it returns information in the rts parameter, which is a pointer to a RtSAPstart structure.

```
struct RtSAPstart {
    int     rts_sd;

    int     rts_mode;
#define RTS_MONOLOGUE   0
#define RTS_TWA         1

    int     rts_turn;
#define RTS_INITIATOR   0
#define RTS_RESPONDER   1

    PE      rts_data;

    struct AcSAPstart rts_start;
};
```

The elements of this structure are:

`rts_sd`: the association-descriptor to be used to reference this association;

`rts_mode`: the dialogue mode (either monologue or two-way alternate),

`rts_turn`: the owner of the turn initially;

`rts_data`: any initial data (this is a pointer to a `PElement` structure, which is fully explained in Chapter 5); and,

`rts_start`: a `AcSAPstart` structure (consult page 26).

Note that the `rts_data` element is allocated via *malloc*(3) and should be released by using the `RTSFREE` macro when no longer referenced. The `RTSFREE` macro behaves as if it was defined as:

```
void    RTSFREE (rts)
struct RtSAPstart *rts;
```

The macro frees only the data allocated by `RtInit`, and not the `RtSAPstart` structure itself. Further, `RTSFREE` should be called only if the call to the `RtInit` routine returned `OK`.

If the call to `RtInit` is not successful, then a RT-P-ABORT.INDICATION event is simulated, and the relevant information is returned in an encoded `RtSAPindication` structure (discussed in Section 4.2 on page 95).

After examining the information returned by `RtInit` on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the association. For either response, the responder should use the `RtOpenResponse` routine (which corresponds to the RT-OPEN.RESPONSE action).

```
int     RtOpenResponse (sd, status, context, respondtitle,
        respondaddr, ctxlist, defctxresult, data, rti)
int     sd,
        status;
OID     context;
AEI     respondtitle;
struct PSAPaddr *respondaddr;
struct PSAPctxlist *ctxlist;
int     defctxresult;
PE      data;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`status`: the acceptance indicator (either `ACS_ACCEPT` if the association is to be accepted, or one of the user-initiated rejection codes listed in Table 2.1 on page 29);

`data`: any initial data (regardless of whether the association is to be accepted); and,

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

The remaining parameters are for the association control service, consult the description of the `AcAssocResponse` routine on page 30.

If the call to `RtOpenResponse` is successful, and the `status` parameter was set to `ACS_ACCEPT`, then association establishment has now been completed. If the call was successful, but the `status` parameter was not `ACS_ACCEPT`, then the association has been rejected and the responder may exit. Otherwise, if the call failed and the reason is "fatal", then the association is lost.

Client Initialization

A program wishing to associate itself with another user of reliable transfer services calls the `RtOpenRequest` routine, which corresponds to the RT-OPEN.REQUEST action.

```
int     RtOpenRequest (mode, turn, context, callingtitle,
        calledtitle, callingaddr, calledaddr, ctxlist,
        defctxname, data, qos, rtc, rti)
int     mode,
        turn;
AEI     callingtitle,
        calledtitle;
OID     context;
struct PSAPaddr *callingaddr,
                *calledaddr;
int     single;
```

```
struct PSAPctxlist *ctxlist;
OID     defctxname;
PE      data;
struct QOStype *qos;
struct RtSAPconnect *rtc;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`mode`: the dialogue mode;

`turn`: who gets the initial turn;

`data`: any initial data;

`qos`: the quality of service on the connection (see Section 4.6.2 in *Volume Two*);

`rtc`: a pointer to a `RtSAPconnect` structure, which is updated only if the call succeeds; and,

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

The remaining parameters are for the association control service, consult the description of the `AcAssocRequest` routine on page 32. If the call to `RtOpenRequest` is successful (this corresponds to a RT-OPEN.CONFIRMATION event), it returns information in the `rtc` parameter, which is a pointer to a `RtSAPconnect` structure.

```
struct RtSAPconnect {
    int     rtc_sd;

    int     rtc_result;

    PE      rtc_data;

    struct AcSAPconnect rtc_connect;
};
```

The elements of this structure are:

`rtc_sd`: the association-descriptor to be used to reference this association;

`rtc_result`: the association response;

`rtc_data`: any initial data (regardless of whether the association was accepted); and,

`rtc_connect`: a `AcSAPconnect` structure (consult page 33).

If the call to `RtOpenRequest` is successful, and the `rtc_result` element is set to `RTS_ACCEPT`, then association establishment has completed. If the call is successful, but the `rtc_result` element is not `RTS_ACCEPT`, then the association attempt has been rejected; consult Table 4.1 to determine the reason for the reject. Otherwise, if the call fails then the association is not established and the `RtSAPabort` structure of the `RtSAPindication` discriminated union has been updated.

Note that the `rtc_data` element is allocated via *malloc*(3) and should be released using the `RTCFREE` macro when no longer referenced. The `RTCFREE` macro behaves as if it was defined as:

```
void    RTCFREE (rtc)
struct RtSAPconnect *rtc;
```

The macro frees only the data allocated by `RtOpenRequest`, and not the `RtSAPconnect` structure itself. Further, `RTCFREE` should be called only if the call to the `RtOpenRequest` routine returned `OK`.

4.1.2 Association Release

The `RtCloseRequest` routine is used to request the release of an association, and corresponds to a RT-CLOSE.REQUEST action.

```
int     RtCloseRequest (sd, reason, data, acr, rti)
int     sd,
        reason;
PE      data;
struct AcSAPrelease *acr;
struct RtSAPindication *rti;
```

The parameters to this procedure:

`sd`: the association-descriptor;

`reason`: the reason why the association should be released, one of:

Value	Reason
ACF_NORMAL	normal
ACF_URGENT	urgent
ACF_UNDEFINED	undefined

`data`: any final data;

`acr`: a pointer to a `AcSAPrelease` structure, which is updated only if the call succeeds; and,

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

If the call to `RtCloseRequest` is successful, then this corresponds to a RT-CLOSE.CONFIRMATION event, and it returns information in the `acr` parameter, which is a pointer to a `AcSAPrelease` structure (consult page 37).

If the call to `RtCloseRequest` is successful, then the association has been released. Otherwise the `RtSAPabort` element of the `RtSAPindication` parameter `rti` contains the reason for failure.

Upon receiving a RT-CLOSE.INDICATION event, the user is required to generate a RT-CLOSE.RESPONSE action using the `RtCloseResponse` routine.

```
int     RtCloseResponse (sd, reason, data, rti)
int     sd,
        reason;
PE      data;
struct RtSAPindication *rti;
```

The parameters to this procedure:

`sd`: the association-descriptor;

`reason`: the reason for the indicator, one of:

Value	Reason
ACR_NORMAL	normal
ACR_NOTFINISHED	not finished
ACR_UNDEFINED	undefined

`data`: any final data; and,

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

If the call to `RtCloseResponse` is successful, then the association has been released. If the call was successful, but the `reason` parameter was not `ACR_NORMAL`, then the association remains established.

4.1.3 Association Abort

To unilaterally abort an association, the routine `RtUAbortRequest` routine is used which corresponds to the RT-U-ABORT.REQUEST action.

```
int     RtUAbortRequest (sd, data, rti)
int     sd;
PE      data;
struct RtSAPindication *rti;
```

The parameters to this procedure:

`sd`: the association-descriptor;

`data`: any abort data; and,

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

If the call to `RtUAbortRequest` is successful, then the association is immediately relesaed, and any data queued for the association may be lost.

4.2 Reliable Transfer

NOTE:	Users should also consult Sectin 4.2.6 on page 104 for optional routines to make RTS behave more sanely.

Once the association has been established, an association-descriptor is used to reference the association. This is usually the first parameter given to any of the remaining routines in the *librtsap*(3n) library. Further, the last parameter is usually a pointer to a `RtSAPindication` structure. If a call to one of these routines fails, then the structure is updated.

```
struct RtSAPindication {
    int     rti_type;
#define RTI_TURN        0x00
#define RTI_TRANSFER    0x01
#define RTI_ABORT       0x02
#define RTI_CLOSE       0x03 (X.410 CLOSE)
#define RTI_FINISH      0x04 (RT-CLOSE)

    union {
        struct RrSAPturn rti_un_turn;
        struct RtSAPtransfer rti_un_transfer;
        struct RtSAPabort rti_un_abort;
        struct RtSAPclose rti_un_close;
        struct AcSAPfinish rti_un_finish
    }   rti_un;
#define rti_turn        rti_un.rti_un_turn
#define rti_transfer    rti_un.rti_un_transfer
#define rti_abort       rti_un.rti_un_abort
#define rti_close       rti_un.rti_un_close
#define rti_finish      rti_un.rti_un_finish
};
```

As shown, this structure is really a discriminated union (a structure with a tag element followed by a union). Hence, on a failure return, one first coerces a pointer to the `RtSAPabort` structure contained therein, and then consults the elements of that structure.

```
struct RtSAPabort {
```

```
    int     rta_peer;

    int     rta_reason;

    PE      rta_udata;

#define RTA_SIZE        512
    int     rta_cc;
    char    rta_data[RTA_SIZE];
};
```

The elements of a `RtSAPabort` structure are:

`rta_peer`: if set, indicates a user-initiated abort (a RT-U-ABORT.INDICATION event); if not set, indicates a provider-initiated abort (a RT-P-ABORT.INDICATION event);

`rta_reason`: the reason for the abort (codes are listed in Table 4.1);

`rta_udata`: optionally, data associated with the user-initiated abort; and,

`rta_data/rta_cc`: a diagnostic string from the provider.

Note that the `rta_udata` element is allocated via *malloc*(3) and should be released using the `RTAFREE` macro when no longer referenced. The `RTAFREE` macro behaves as if it was defined as:

```
void    RTAFREE (rta)
struct RtSAPabort *rta;
```

The macro frees only the data allocated in the `RtSAPabort` structure and not the structure itself.

On a failure return, if the `rta_reason` element of the `RtSAPabort` structure is associated with a fatal error, the the association is released. The `RTS_FATAL` macro can be used to determine this.

```
int     RTS_FATAL (r)
int     r;
```

Provider-Initiated Aborts (fatal)	
`RTS_ADDRESS`	Address unknown
`RTS_REFUSED`	Connect request refused on this network connection
`RTS_SESSION`	Session disconnect
`RTS_PROTOCOL`	Protocol error
`RTS_CONGEST`	Congestion at RtSAP
`RTS_REMOTE`	Remote system problem
`RTS_PRESENTATION`	Presentation disconnect
`RTS_ACS`	ACS disconnect
`RTS_ABORTED`	Peer aborted association
User-Initiated Rejects (fatal)	
`RTS_BUSY`	Busy
`RTS_RECOVER`	Cannot recover
`RTS_VALIDATE`	Validation failure
`RTS_MODE`	Unacceptable dialogue mode
`RTS_REJECT`	Rejected by responder
Interface Errors (non-fatal)	
`RTS_PARAMETER`	Invalid parameter
`RTS_OPERATION`	Invalid operation
`RTS_TIMER`	Timer expired
`RTS_WAITING`	Indications waiting
`RTS_TRANSFER`	Transfer failure

Table 4.1: RtSAP Failure Codes

For protocol purists, the RTS_OFFICIAL macro can be used to determine if the error is an "official" error as defined by the specification, or an "unofficial" error used by the implementation.

```
int     RTS_OFFICIAL (r)
int     r;
```

4.2.1 Sending Data

When the user has the turn, it can use the RtTransferRequest routine (this corresponds to the RT-TRANSFER.REQUEST action) to request the reliable transfer of a data structure.

```
int     RtTransferRequest (sd, data, secs, rti)
int     sd;
PE      data;
int     secs;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

sd: the association-descriptor;

data: the data to be transferred;

secs: the amount of time, in seconds, permitted to transfer the data (use the manifest constant NOTOK if time is unimportant); and,

rti: a pointer to a RtSAPindication structure, which is updated only if the call fails.

If RtTransferRequest succeeds, then the data has been reliably transferred to the other user. Otherwise the RtSAPabort structure contained in the RtSAPindication parameter rti contains the reason for failure.

4.2.2 Receiving Data

The RtWaitRequest routine is used to wait for an event (usually incoming data) to occur.

```
int     RtWaitRequest (sd, secs, rti)
int     sd;
int     secs;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

sd: the association-descriptor;

secs: the maximum number of seconds to wait for the event (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,

rti: a pointer to a RtSAPindication structure, which is always updated.

Unlike the other routines in the *librtsap*(3n) library, the RtWaitRequest routine returns one of three values: NOTOK (on failure), OK (on reading data) or DONE (when something else happens).

If the call to RtWaitRequest returns the manifest constant NOTOK, then the RtSAPabort structure contained in the RtSAPindication parameter rti contains the reason for the failure.

Otherwise if the call to RoWaitRequest returns the manifest constant OK, then data has arrived. This event is encoded in the rti parameter, depending on the value of the rti_type element. Currently, when RoWaitRequest returns OK, the rti_type element is set to RTI_TRANSFER, which indicates that a RT-TRANSFER.REQUEST event has occurred.

Otherwise if the call to RtWaitRequest returns the manifest constant DONE, then some event other than data arriving has occurred. This event is also encoded in the rti parameter, depending on the value of the rti_type element. Currently, when RoWaitRequest returns DONE, the rti_type element is set to one of two values:

Value	Event
RTI_TURN	RT-TURN-PLEASE.INDICATION
RTI_TURN	RT-TURN-GIVE.INDICATION
RTI_CLOSE	X.410 CLOSE.INDICATION (for old-style associations)
RTI_FINISH	RT-CLOSE.INDICATION

Transfer Indication

When an RT-TRANSFER.INDICATION event occurs, the `rti_type` field of the `rti` parameter contains the value `RTI_TRANSFER`, and a `RtSAPtransfer` structure is contained inside the `rti` parameter.

```
struct RtSAPtransfer {
    PE      rtt_data;
};
```

The elements of this structure are:

`rtt_data`: the data received.

Note that the `rtt_data` element is allocated via *malloc*(3) and should be released using the `RTTFREE` macro when no longer referenced. The `RTTFREE` macro behaves as if it was defined as:

```
void    RTTFREE (rtt)
struct RtSAPtransfer *rtt;
```

The macro frees only the data allocated in the `RtSAPtransfer` structure and not the structure itself.

Turn Indication

When an RT-TURN-GIVE.INDICATION or RT-TURN-PLEASE.INDICATION events occur, the `rti_type` field of the `rti` parameter contains the value `RTI_TURN`, and a `RtSAPturn` structure is contained inside the `rti` parameter.

```
struct RtSAPturn {
    int     rtu_please;

    int     rtu_priority;
};
```

The elements of this structure are:

`rtu_please`: if set, indicates that a RT-PLEASE-TURN.INDICATION event has occurred; if not set, indicates that a RT-GIVE-TURN.INDICATION event has occurred; and,

`rtu_priority`: the priority at which the turn is requested (meaningful only if `rtu_please` is set).

Close Indication

When an X.410 CLOSE.INDICATION event occurs, the rti_type field of the rti parameter contains the value RTI_CLOSE, and a RtSAPclose structure is contained inside the rti parameter.

```
struct RtSAPclose {
    int     rtc_dummy;
};
```

Finish Indication

When an RT-CLOSE.INDICATION event occurs, the rti_type field of the rti parameter contains the value RTI_FINISH, and a AcSAPfinish structure is contained inside the rti parameter.

4.2.3 Managing the Turn

The user with the turn is permitted to send data to the other user. To request the turn, one invokes the RtPTurnRequest routine, which corresponds to the RT-PLEASE-TURN.REQUEST.

```
int     RtPTurnRequest (sd, priority, rti)
int     sd;
int     priority;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

sd: the association-descriptor;

priority: the priority at which the turn is requested (this is defined by each application); and,

rti: a pointer to a RtSAPindication structure, which is updated only if the call fails.

If the call to the RtPTurnRequest routine succeeds, then the turn has been requested of the remote user. Otherwise the RtSAPabort structure contained in the RtSAPindication parameter rti contains the reason for failure.

To relinquish the turn, one invokes the RtGTurnRequest routine, which corresponds to the RT-GIVE-TURN.REQUEST.

```
int     RtGTurnRequest (sd, rti)
int     sd;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd`: the association-descriptor; and,

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

If the call to the `RtGTurnRequest` routine succeeds, then the turn has been relinquished to the remote user. Otherwise the `RtSAPabort` structure contained in the `RtSAPindication` parameter `rti` contains the reason for failure.

4.2.4 Asynchronous Event Handling

The events discussed thus far have been synchronous in nature. Some users of the reliable transfer service may wish an asynchronous interface. The `RtSetIndications` routine is used to change the service associated with an association-descriptor to an asynchronous interface.

```
int   RtSetIndications (sd, indication, rti)
int   sd;
IFP   indication;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`indication`: the address of an event-handler routine to be invoked when an event occurs; and,

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

If the service is to be made asynchronous, then `indication` is specified; otherwise, if the service is to be made synchronous, it is not specified (use the manifest constant `NULLIFP`). The most likely reason for the call failing is `RTS_WAITING`, which indicates that an event is waiting for the user.

When an event occurs, the event-handler routine is invoked with two parameters:

```
(*handler) (sd, rti);
int     sd;
struct RtSAPindication *rti;
```

The parameters are:

`sd`: the association-descriptor; and,

`rti`: a pointer to a `RtSAPindication` structure encoding the event.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released with the appropriate macro (either `RTTFREE` or `RTPFREE`) when no longer needed.

When an event-handler is invoked, future invocations of the event-hander are blocked until it returns. The return value of the event-handler is ignored. Further, during the execution of a synchronous call to the library, the event-handler will be blocked from being invoked.

NOTE:	The *librtsap*(3n) library uses the SIGEMT signal to provide these services. Programs using asynchronous association-descriptors should NOT use SIGEMT for other purposes.

4.2.5 Synchronous Event Multiplexing

A user of the reliable transfer service may wish to manage multiple association-descriptors simultaneously; the routine `RtSelectMask` is provided for this purpose. This routine updates a file-descriptor mask and associated counter for use with `xselect` (consult Section 2.4), as assocation-descriptors are file-descriptors.

```
int     RtSelectMask (sd, mask, nfds, rti)
int     sd;
fd_set *mask,
int    *nfds;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`mask`: a pointer to a file-descriptor mask meaningful to `xselect`;

`nfds`: a pointer to an integer-valued location meaningful to `xselect`; and,

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

If the call is successful, then the `mask` and `nfds` parameters can be used as arguments to `xselect`. The most likely reason for the call failing is `RTS_WAITING`, which indicates that an event is waiting for the user.

If `xselect` indicates that the association-descriptor is ready for reading, `RtWaitRequest` should be called with the `secs` parameter equal to `OK`. If the network activity does not constitute an entire event for the user, then `RtWaitRequest` will return `NOTOK` with error code `RTS_TIMER`.

4.2.6 Reliable Transfer (revisited)

The mechanism provided by `RtTransferRequest` may not be useful for applications which have large amounts of data to transfer. In most cases, it is preferable to transfer data incrementally. To provide for this functionality, the routine `RtSetDownTrans` is provided:

```
int     RtSetDownTrans (sd, fnx, rti)
int     sd;
IFP     fnx;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd`: the association descriptor;

`fnx`: the address of an event-handler routine to be invoked when data is needed; and,

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

If the `fnx` parameter is some value other than the manifest constant `NULLIFP`, then upon successful return from `RtSetDownTrans`, the behavior of the routine `RtTransferRequest` is altered.

1. The `data` parameter to `RtTransferRequest` may be `NULLPE`.

2. In this case, the event-handler routine `fnx` will be invoked in order to retrieve a portion of the data to be transferred:

   ```
   result = (*fnx) (sd, base, len, size, ack, ssn, rti);
   int     sd;
   char  **base;
   int    *len,
           size;
   long    ack,
           ssn;
   struct RtSAPindication *rti;
   ```

   where `sd` is the association-descriptor which was given to the routine `RtTransferRequest`.

3. If the `base` parameter has the value `NULLVP`, then a RT-PLEASE.INDICATION event is being signaled, and the `size` parameter has the value of the priority associated with the event.

4. Otherwise, the `base` and `len` parameters point to a pointer/length pair which should be set by the event handler to upto `size` octets. The event handler is responsible for any memory allocation (e.g., allocating a buffer of `size` octets and then assigning the address of the buffer to `*base`). Once a buffer is chosen, the event handler should read upto `size` octets into the buffer and set `*len` to the number of octets actually read. The `ssn` and `ack` parameters given the values of the last synchronization point requested and acknolwedged (how this information should be used is unknown at this time). If the value assigned to `*len` is zero, then this indicates that all data has been read and the transfer should be completed.

5. If the value of the `size` is zero, then this indicates that the provider could not negotiate a incremental transfer and the event handler should allocate a buffer of arbitrary size, read all of the data to be transferred into that one buffer, and then update `*base` and `*len` accordingly.

6. If the event handler encounters an error, it should return the manifest constant `NOTOK` (otherwise, it should return `OK`). If an error is signaled,

the event handler should update the `rti` structure accordingly. The easiest way to do this is:

```
return rtsaplose (rti, RTS_TRANSFER, NULLCP, "text");
```

If the event is RT-PLEASE.REQUEST, then signaling an error results in the provider generating an S-ACTIVITY-INTERRUPT.REQUEST; otherwise when an error is signaled, the provider will generate an S-ACTIVITY-DISCARD.REQUEST to abort the transfer.

For similar reasons, the mechanism employed by `RtWaitRequest` may not be useful for applications which have large amounts of data to transfer. Again, in most cases it is preferable to transfer data incrementally. To provide for this functionality, the routine `RtSetUpTrans` is provided:

```
int     RtSetUpTrans (sd, fnx, rti)
int     sd;
IFP     fnx;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd`: the association descriptor;

`fnx`: the address of an event-handler routine to be invoked when data has been received; and,

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

If the `fnx` parameter is some value other than the manifest constant `NULLIFP`, then upon successful return from `RtSetUpTrans`, the behavior of the routine `RtWaitRequest` is altered when it returns an RT-TRANSFER.INDICATION (the `rti_type` field of the `rti` parameter contains the value `RTI_TRANSFER`, and a `RtSAPtransfer` structure is contained inside the `rti` parameter). No data is returned by `RtWaitRequest`, rather as data is received, the event routine is invoked:

1. When data is received, the event-handler routine `fnx` will be invoked in order to store a portion of the data being transferred:

```
result = (*fnx) (sd, event, addr, rti);
int     sd;
int     event;
caddr_t addr;
struct RtSAPindication *rti;
```

where `sd` is the association-descriptor which was given to the routine `RtWaitRequest`.

2. If the `event` parameter has the value `SI_DATA`, then the `addr` parameter is really a pointer to a `struct qbuf` structure, and the event handler should traverse the qbuf writing out the data found therein.

3. If the `event` parameter has the value `SI_SYNC`, then the `addr` parameter is really a pointer to a `struct SSAPsync` structure, and the event handler should note the information contained therein.Currently, this will only occur when a S-MINOR-SYNC.INDICATION event is signaled.

4. If the `event` parameter has the value `SI_ACTIVITY`, then the `addr` parameter is really a pointer to a `struct SSAPactivity` structure, and the event handler should note the information contained therein. Currently, there are four events that are signaled:

   - S-ACTIVITY-START.INDICATION which indicates that a transfer is about to begin;
   - S-ACTIVITY-END.INDICATION which indicates that a transfer is about to complete; and,
   - S-ACTIVITY-INTERRUPT.INDICATION and S-ACTIVITY-DISCARD.INDICATION which indicate that an activity is either suspended or aborted.

5. If the `event` parameter has the value `SI_REPORT`, then the `addr` parameter is really a pointer to a `struct SSAPreport` structure, and the event handler should note the information contained therein. Currently, this will only occur when a S-U-EXCEPTION-REPORT.INDICATION event is signaled.

6. If the event handler encounters an error, it should return the manifest constant `NOTOK` (otherwise, it should return `OK`). If an error is signaled,

the event handler should update the `rti` structure accordingly. The easiest way to do this is:

```
return rtsaplose (rti, RTS_TRANSFER, NULLCP, "text");
```

If the event is S-ACTIVITY-INTERRUPT.INDICATION, S-ACTIVITY-INTERRUPT.INDICATION, or S-ACTIVITY-INTERRUPT.INDICATION, then signaling an error is ignorred by the provider; otherwise, when an error is signaled, the provider will generate an S-U-EXCEPTION-REPORT.REQUEST to abort the transfer.

4.3 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `rti` parameter given to the routine. The `rti_abort` element of the `RtSAPindication` structure encodes the reason for the failure. One coerces a pointer to a `RtSAPabort` structure, and consults the `rta_reason` element of this latter structure. This element can be given as a parameter to the routine `RtErrString` which returns a null-terminated diagnostic string.

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

4.4 Compiling and Loading

Programs using the *librtsap*(3n) library should include `<isode/rtsap.h>`, which automatically includes the header file `<isode/psap.h>` described in Chapter 5. The programs should also be loaded with `-lisode`.

4.5 An Example

Let's consider how one might construct a generic server that uses reliable transfer services to establish an association, but then uses remote operation services to communicate with its peer. This entity will use a synchronous interface.

There are two parts to the program: initialization and the request/reply loop. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure 4.1, the initialization steps for the generic responder, including the outer *C* wrapper, is shown. First, the RtSAP state is re-captured by calling `RtInit`. If this succeeds, then the association is authenticated and any command line arguments are parsed. Assuming that the responder is satisfied with the proposed association, it calls `RtOpenResponse` to accept the association. The `RoSetService` routine is called to set the underlying service to be used for remote operations. Finally, the main request/reply loop is realized. The responder calls `RoWaitRequest` to get the next event, and then calls `ros_indication` to decode that event.

Figure 3.2 on page 77 contains the definition of the `ros_indication` routine and associated routines. Since the reliable transfer service was used to establish the association, a different closing handler must be used. This is shown in Figure 4.2.

```
#include <stdio.h>
#include <isode/rtsap.h>


main (argc, argv, envp)
int     argc;
char  **argv,
      **envp;
{
    int     result,
            sd;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;
    struct RtSAPstart   rtss;
    register struct RtSAPstart *rts = &rtss;
    struct AcSAPstart *acs = &rts -> rts_start;
    strut PSAPstart *ps = &acs -> acs_start;
    struct RtSAPindication  rtis;
    register struct RtSAPindication *rti = &rtis;
    register struct RtSAPabort    *rta = &rti -> rti_abort;

    if (RtInit (argc, argv, rts, rti) == NOTOK)
        error ("initialization fails:  %s", RtErrString (rta -> rta_reason));

    sd = rts -> rts_sd;
    RTSFREE (rts);

/* read command line arguments here... */

    if (RtOpenResponse (sd, ACS_ACCEPT, NULLOID, NULLAEI,
                &ps -> ps_called, NULLPC, ps -> ps_defctxresult,
                NULLPE, rti) == NOTOK)
        error ("RT-OPEN.RESPONSE: %s", RtErrString (rta -> rta_reason));

    if (RoSetService (sd, RoRtService, roi) == NOTOK)
        error ("RoSetService:  %s", RoErrString (rop -> rop_reason));

    for (;;)
        switch (result = RoWaitRequest (sd, NOTOK, roi)) {
            case NOTOK:
            case OK:
            case DONE:
                ros_indication (sd, roi);
                break;

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

Figure 4.1: Initializing the generic RTS responder

```
...

static int ros_finish (sd, rof)
int     sd;
struct AcSAPfinish *acf;
{
    struct RtSAPindication rtis;
    register struct RtSAPabort   *rta = &rtis.rti_abort;

    if (RtCloseResponse (sd, ACR_NORMAL, NULLPE, &rtis) == NOTOK)
        error ("RT-CLOSE.RESPONSE: %s", RtErrString (rta -> rta_reason));

    ACFFREE (acf);

    exit (0);
}
```

Figure 4.2: Finalizing the generic RTS responder

4.6 For Further Reading

[CCI84b] is the CCITT recommendation describing the reliable transfer service. The draft CCITT work which assumes the use of new-style associations is defined in [CCI88a] and [CCI88b]. Similarly, the corresponding ISO work is [ISO88c] and [ISO88d].

Part III

Data Services

Chapter 5

Encoding of Data-Structures

The *libpsap*(3) library implements presentation syntax abstractions for the machine-independent exchange of data structures. There are two objects which are manipulated: *presentation elements*, which represent a particular, arbitrarily complex, data structure; and, *presentation streams*, which represent an I/O path of these data structures.

5.1 Presentation Streams

A presentation stream is an object, similar to a `FILE` object in *stdio*(3s), which is used to read and write *presentation elements*. The `PStream` structure contains several elements, only the most interesting are described here:

`ps_errno`: the latest error to occur on the stream (codes are listed in Table 5.1);

`ps_addr`: the bottom-specific pointer;

`ps_primeP`: the bottom-specific prime routine;

`ps_readP`: the bottom-specific read routine;

`ps_writeP`: the bottom-specific write routine; and,

`ps_flushP`: the bottom-specific flush routine; and,

`ps_closeP`: the bottom-specific close routine.

PS_ERR_NONE	No error
PS_ERR_OVERID	Overflow in ID
PS_ERR_OVERLEN	Overflow in length
PS_ERR_NMEM	Out of memory
PS_ERR_EOF	End of file
PS_ERR_EOFID	End of file reading extended ID
PS_ERR_EOFLEN	End of file reading extended length
PS_ERR_LEN	Length mismatch
PS_ERR_TRNC	Truncated
PS_ERR_INDF	Indefinite length in primitive form
PS_ERR_IO	I/O error
PS_ERR_XXX	Internal error code

Table 5.1: Presentation Stream Failure Codes

The typedef `PS` is a pointer to an `PStream` structure. The `ps_errno` element of the `PStream` structure can be given as a parameter to the routine `ps_error` which returns a null-terminated diagnostic string.

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

5.1.1 Creating a Stream

A `PStream` structure is created by calling the procedure `ps_alloc`, with the address of an integer-valued initialization routine.

```
PS      ps_alloc (init)
int   (*init) ();
```

The `ps_alloc` routine allocates a new structure and then calls the initialization routine passed as a parameter, which should initialize the elements of the structure. The initialization routine should return the manifest constant `OK` if all went well; otherwise it should return the manifest constant `NOTOK` on error, which results in `ps_alloc` freeing the newly allocated structure and then returning the value `NULLPS`.

Several standard initialization routines are available:

`std_open`: for presentation streams connected to *stdio*(3s) `FILE` objects;

```
int     std_open (ps)
PS      ps;
```

`str_open`: for presentation streams connected to string objects.

```
int     str_open (ps)
PS      ps;
```

`fdx_open`: for presentation streams connected to full-duplex file descriptors.

```
int     fdx_open (ps)
PS      ps;
```

and,

`dg_open`: for presentation streams connected to datagram-based file descriptors.

```
int     dg_open (ps)
PS      ps;
```

Presentation streams which have been initialized by these routines will automatically allocate additional resources when necessary, to the limits allowed by the operating system (e.g., repeated writes to a presentation stream connected to a string object will result in additional memory being allocated). In the current implementation, presentation streams which have been initialized by these routines are uni-directional. That is, the presentation stream may be used for reading, or writing, but not both.

After `ps_alloc` successfully returns, final initialization is performed by calling a setup routine, usually either

`std_setup`: for file objects.

```
int     std_setup (ps, fp)
PS      ps;
FILE   *fp;
```

The parameters to this procedure are:

ps: the presentation stream; and,

fp: a pointer to the FILE object to be bound to the presentation stream.

str_setup: for string objects.

```
int     str_setup (ps, cp, cc, inline)
PS      ps;
char   *cp;
int     cc,
inline;
```

The parameters to this procedure are:

ps: the presentation stream;

cp: a pointer to the string to be bound to the presentation stream (use the manifest constant NULLCP if the stream is to be written);

cc: the length of the string; and,

inline: a magic argument, always use 0 unless you know what you're doing.

fdx_setup: for full-duplex file-descriptor objects.

```
int     fdx_setup (ps, fd)
PS      ps;
int     fd;
```

The parameters to this procedure are:

ps: the presentation stream; and,

fd: the file-descriptor.

dg_setup: for datagram objects.

```
int     dg_setup (ps, fd, size, rfx, wfx)
PS      ps;
int     fd,
        size;
```

```
IFP     rfx,
        wfx;
```

The parameters to this procedure are:

`ps`: the presentation stream;

`fd`: the file-descriptor;

`size`: the maximum datagram size; and,

`rfx/wfx`: routines to read and write datagrams.

After the setup routine successfully returns (by returning the manifest constant `OK`), the presentation stream is ready for reading or writing.

5.1.2 Stream I/O

Low-level I/O is done from/to the stream by the macros `ps_read` and `ps_write`, which behave as if they were defined as:

```
int     ps_read (ps, data, cc, inline)
PS      ps;
char   *data;
int     cc,
        inline;

int     ps_write (ps, data, cc, inline)
PS      ps;
char   *data;
int     cc,
        inline;
```

The parameters to both of these macros are the same:

`ps`: the presentation stream;

`data`: the address of a character buffer;

`cc`: the number of characters to read/write from/to the buffer; and,

`inline`: a magic argument, always use `0` unless you know what you're doing.

These both call an internal routine, `ps_io`, which switches to the object-specific read or write routine as appropriate. The `ps_io` procedure will call the object-specific routines as many times as required to read/write the full number of `cc` bytes from/to the `data` buffer.

5.1.3 Deleting a Stream

The routine `ps_free` is used to close and deallocate a presentation stream.

```
void    ps_free (ps)
PS      ps;
```

It takes a single parameter, a pointer to the presentation stream to be freed. This routine first calls the routine specified by the `ps_closeP` element in the `PStream` structure (if any). It then frees the structure itself.

5.1.4 Implementing Other Abstractions

Let us briefly consider the internal protocol and uniform interface used in the implementation of presentation streams.

The initialization routine given as an argument to `ps_alloc` typically initializes only the

```
ps_primeP
ps_readP
ps_writeP
ps_flushP
ps_closeP
```

elements in the `PStream` structure.

The setup routine is entirely dependent on the particular object used to realize the I/O abstraction for the presentation stream. In most cases, it allocates a structure of its own and sets the `ps_addr` element to the address of the structure.

The `ps_readP` and `ps_writeP` elements of the `PStream` structure are used by the `ps_io` routine as required for reading and writing.

```
int     ps_io (ps, io, data, cc, inline)
PS      ps;
```

```
int   (*io) ();
char   *data;
int     cc,
```

The parameters to these routines are identical to those for their counterpart macros, ps_read and ps_write, with one exception:

io: the address of an integer-valued function which does the actual reading or writing. It is invoked as:

```
int     n = (*io) (ps, data, len, inline);
```

where a return value of NOTOK indicates an error occurred (the routine should set the ps_errno element of the presentation stream); OK indicates that the end-of-file has been read; and, any other value is the number of bytes actually transferred (which should be greater than 0 but not greater than len).

For some packet-oriented applications, it may be desirable to do a single "read from the network" before reading the components of a presentation element. The routine pointed to by the ps_primeP element of the PStream structure is called by ps2pe (described momentarily) before any parts of the entire presentation element has been read. The routine is invoked as:

```
(*ps -> ps_primeP) (ps, waiting)
PS      ps;
int waiting;
```

The routine should return NOTOK if an error occurred (setting the ps_errno element of the presentation stream in the process); otherwise, if data is already queued and waiting is non-zero, then DONE is returned. otherwise OK is returned. The routine ps_prime may be called to explicitly "prime the pump" associated with a presentation element:

```
int     ps_prime (ps)
PS      ps;
```

This routine returns OK on success, or NOTOK on failure.

In order to improve efficiency, it may be desirable to have ps_writeP buffer output. The routine pointed to by the ps_flushP element of the PStream structure is called by pe2ps (described momentarily) after the entire presentation element has been written. The routine is invoked as:

```
(*ps -> ps_flushP) (ps)
PS      ps;
```

The routine should return NOTOK if an error occurred (setting the ps_errno element of the presentation stream in the process); or OK if everything was fine. The routine ps_flush may be called to explicitly flush any buffers associated with a presentation element:

```
int     ps_flush (ps)
PS      ps;
```

This routine returns OK on success, or NOTOK on failure.

Finally, the ps_closeP element of the PStream structure is used by the ps_free routine to release any resources which the setup routine may have allocated. For example, if the setup routine allocated a structure and set the ps_addr element of the presentation stream to point to that structure, then the function pointed to by the ps_closeP element should free that structure, and, as a matter of good programming practice, set ps_addr to NULL.

5.2 Presentation Stream I/O

The routine ps2pe can be used to read the next presentation element from a presentation stream. This routine returns a pointer to the presentation element or the manifest constant NULLPE on error. (The typedef PE is a pointer to a structure containing a presentation element; presentation elements are described more fully later.)

```
PE      ps2pe (ps)
PS      ps;
```

Similarly, the routine pe2ps can be used to write a presentation element at the end of the presentation stream, returning OK if all went well, or NOTOK otherwise.

```
int     pe2ps (ps, pe)
PS      ps;
PE      pe;
```

On errors with either routine, the `ps_errno` element of the `PStream` structure can be consulted to see what happened (see Table 5.1 on page 116).

When writing to a presentation stream, the variable `ps_len_strategy` controls how lengthy data structures are represented by determining when the "indefinite form" is used to encode the length of the data structure.

```
int     ps_len_strategy;
```

If this variable is equal to `PS_LEN_SPAG` (the default), then the indefinite form is used whenever the length field of the presentation element cannot be represented in one octet. If the value instead is `PS_LEN_INDF`, then the indefinite form is used regardless of the length of the presentation element. Otherwise, if the value is `PS_LEN_LONG`, then the indefinite form is never used.

The routine `ps_get_abs` can be used to determine the total number of octets that will be required to represent the presentation element when written to a presentation stream. This is useful for buffer management purposes.

```
int     ps_get_abs (pe)
PE      pe;
```

5.2.1 Debugging

For debugging purposes, instead of treating a presentation stream as a binary object, the routines `pl2pe` and `pe2pl` can be used.

```
PE      pl2pe (ps)
PS      ps;

int     pe2pl (ps, pe)
PS      ps;
PE      pe;
```

These translate between presentation *lists* and presentation elements. A presentation list is identical to a presentation stream, but instead of using a binary representation, a list uses an ASCII text representation with a simple LISP-like syntax.

5.3 Presentation Elements

A presentation element is an object which is used to represent a data structure in a machine-independent form. The `PElement` structure contains several elements, only the most interesting are described here:

`pe_errno`: the latest error to occur on the presentation element (codes are listed in Table 5.2);

`pe_context`: the presentation context to which the element belongs (consult Section 2.3.1 on page 18 of *Volume Two*);

`pe_class`: the class of this presentation element (i.e., one of *universal*, *application-wide*, *context-specific*, or *private*);

`pe_form`: the form taken by this presentation element (i.e., *primitive*, or *constructed*);

`pe_id`: the class-specific code identifying the type of this presentation element;

`pe_offset`: the offset of this presentation element in a sequence; and,

`pe_next`: a pointer to the next presentation element in a sequence.

As mentioned earlier, the typedef `PE` is a pointer to an `PElement` structure. With the exception of the `pe_errno` element, the elements of the `PElement` structure are largely uninteresting to the user of the *libpsap*(3) library. The element `pe_errno` can be given as a parameter to the routine `pe_error` which returns a null-terminated diagnostic string.

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

Once a presentation stream has been initialized and elements are being read, there are several routines which can be used to translate between the machine-independent representation of the element and machine-specific objects such as integers, strings, and the like. It is extremely important that programs use these routines to perform the translation between objects. They have been carefully coded to present a simple, uniform interface between machine-specifics and the machine-independent encoding protocol.

PE_ERR_NONE	No error
PE_ERR_OVER	Overflow
PE_ERR_NMEM	Out of memory
PE_ERR_BIT	No such bit
PE_ERR_UTCT	Malformed universal timestring
PE_ERR_GENT	Malformed generalized timestring
PE_ERR_MBER	No such member
PE_ERR_PRIM	Not a primitive form
PE_ERR_CONS	Not a constructor form
PE_ERR_TYPE	Class/ID mismatch in constructor
PE_ERR_OID	Malformed object identifier
PE_ERR_BITS	Malformed bitstring

Table 5.2: Presentation Element Failure Codes

5.3.1 Creating an Element

A `PElement` structure is created by calling the procedure `pe_alloc`.

```
PE      pe_alloc (class, form, id)
PElementClass class;
PElementForm form;
PElementID id;
```

This procedure takes three parameters:

`class`: the class of this presentation element. The codes are:

PE_CLASS_UNIV	Universal
PE_CLASS_APPL	Application-wide
PE_CLASS_CONT	Context-specific
PE_CLASS_PRIV	Private-use

`form`: the form of this presentation element The codes are:

PE_FORM_PRIM	Primitive
PE_FORM_CONS	Constructor

`id`: the class-specific code identifying the type of this presentation element (codes for the "universal" class are listed in Table 5.3).

Internal Use	
PE_UNIV_EOC	End-of-contents
Built-in Types	
PE_PRIM_BOOL	Boolean
PE_PRIM_INT	Integer
PE_PRIM_BITS	Bitstring
PE_PRIM_OCTS	Octetstring
PE_PRIM_NULL	Null
PE_PRIM_OID	Object Identifier
PE_PRIM_ODE	Object Descriptor
PE_CONS_EXTN	External
PE_CONS_SEQ	Sequence
PE_CONS_SET	Set
Defined Types	
PE_DEFN_IA5S	IA5 String
PE_DEFN_NUMS	Numeric String
PE_DEFN_PRTS	Printable String
PE_DEFN_T61S	T.61 String
PE_DEFN_VTXS	Videotex String
PE_DEFN_GENT	Generalized Time
PE_DEFN_UTCT	UTC Time
PE_DEFN_GFXS	Graphics String
PE_DEFN_VISS	Visual String
PE_DEFN_GENS	General String

Table 5.3: Presentation Element Identifiers

5.3.2 Deleting an Element

The routine `pe_free` is used to deallocate a presentation element.

```
void    pe_free (pe)
PE      pe;
```

NOTE:	When using `pe_free` on a presentation element, care must be taken to remove any references to that presentation element in other structures. For example, if you have a sequence containing a sequence, and you free the child sequence, be sure to zero-out the parent's pointer to the child, otherwise subsequent calls using the parent will go romping through hyperspace. See `pe_extract` and `pe_expunge` below.

5.3.3 Primitive Manipulation of Elements

Two presentation elements can be compared with `pe_cmp`, which takes two pointers to `PElement` structures as arguments, and returns `0` if the two structures are identical, `1` otherwise.

```
int     pe_cmp (p, q)
PE      p,
        q;
```

Further, an presentation element can be duplicated with `pe_cpy`. This routine takes a pointer to an `PElement` structure as an argument, and returns a pointer to a new `PElement` structure, or `NULLPE` on error.

```
PE      pe_cpy (pe)
PE      pe;
```

If a presentation element, `pe`, has a descendant `r` (which appears below `pe` exactly once), then `pe_extract` can be used to destroy the relationship between the two presentation elements without freeing either element.

```
int     pe_extract (pe, r)
PE      pe,
        r;
```

This routine returns non-zero if `r` was descended from `pe`, otherwise it returns zero. The `pe_expunge` routine is similar to `pe_extract`, except that it always deallocates the parent element and returns the child element found, if any.

```
PE      pe_expunge (pe, r)
PE      pe,
        r;
```

These two routines are provided primarily for efficiency considerations. They are extremely fragile. Do not use them unless you know what you're doing.

Finally, the `pe_pullup` routine can be used to "pull-up" a constructor type into a primitive.

```
int     pe_pullup (pe)
PE      pe;
```

This routine returns `OK` on success; on failure it turns `NOTOK`, usually due to a memory allocation failure.

5.4 Presentation Element Transformations

We now discuss how machine-specific objects can be encoded in a machine-independent fashion and vice-versa. Routines of the form `XXX2prim` all return a pointer to a presentation element on success, or `NULLPE` on failure (usually due to a failure in the memory allocator). Routines of the form `prim2XXX` all return an `XXX`-valued object on success, or `NOTOK` or `NULLxxx` on failure (depending on the object the routine normally returns). On failure, the `pe_errno` element of the presentation element can be examined to determine the reason for failure.

5.4.1 Boolean

A *boolean* is an integer taking on the values `0` or `1`. The routine `prim2flag` takes a pointer to an presentation element and returns the boolean value encoded therein.

```
int     prim2flag (pe)
PE      pe;
```

The routine bool2prim is a macro which performs the inverse operation. It behaves as if it was defined as:

```
PE      bool2prim (b)
int     b;
```

In actuality, bool2prim calls the routine flag2prim which builds a presentation element containing the boolean value given and sets the pe_class and pe_id fields of the presentation element to the desired values.

```
PE      flag2prim (b, class, id)
int     b;
PElementClass class;
PElementID id;
```

5.4.2 Integer

An *integer* is a signed-quantity, whose precision is specific to a particular host. The routine prim2num takes a pointer to a presentation element and returns the integer value encoded therein (if the value is NOTOK, check the pe_errno element of the PElement structure to see if there really was an error).

```
int     prim2num (pe)
PE      pe;
```

The routine int2prim is a macro which performs the inverse operation. It behaves as if it was defined as:

```
PE      int2prim (i)
int     i;
```

In actuality, int2prim calls the routine num2prim which builds a presentation element containing the numeric value given and sets the pe_class and pe_id fields of the presentation element to the desired values.

```
PE      num2prim (i, class, id)
int     i;
PElementClass class;
PElementID id;
```

5.4.3 Octetstring

An *octetstring* is a pair of values: a character pointer and an integer length. The pointer addresses the first octet in the string (which need not be null-terminated), the length indicates how many octets are in the string. The routine `prim2str` takes a pointer to a presentation element and allocates a new string (using the *malloc*(3) routine) containing the value encoded therein.

```
char   *prim2str (pe, len)
PE      pe;
int    *len;
```

The routine `oct2prim` is a macro which performs the inverse operation, copying the original string (and not de-allocating it). It behaves as if it was defined as:

```
PE      oct2prim (s, len)
char   *s;
int     len;
```

In actuality, `oct2prim` calls the routine `str2prim` which builds a presentation element containing the string value given and sets the `pe_class` and `pe_id` fields of the presentation element to the desired values.

```
PE      str2prim (s, len, class, id)
char   *s;
int     len;
PElementClass class;
PElementID id;
```

In addition to `oct2prim`, there are several macros which manipulate octetstrings, setting the class and id of the presentation element to the appropriate value:

Macro	String
`ia5s2prim`	IA5 string
`nums2prim`	Numeric string
`prts2prim`	Printable string
`t61s2prim`	T.61 string
`vtxs2prim`	Videotex string
`gfxs2prim`	Graphics string
`viss2prim`	Visible string
`gens2prim`	General string
`chrs2prim`	Character string
`ode2prim`	Object descriptor

Each of these macros are defined in a similar manner to the `oct2prim` macro.

5.4.4 Octetstrings revisited

For efficiency reasons, it is often desirable to represent an octetstring using a `qbuf` structure. Although the format of a `qbuf` is described in Section 4.3.2 on page 119 of *Volume Two*, the *libpsap*(3) library provides several routines for ease of manipulation.

The routine `qb2prim` takes a linked-list of `qbuf`s and builds a presentation element with the `pe_class` and `pe_id` fields of the presentation element set to the desired values:

```
PE      qb2prim (qb, class, id)
struct qbuf *qb;
PElementClass class;
PElementID id;
```

NOTE: The presentation element returned by `qb2prim` contains pointers into the linked-list of `qbuf`s. Therefore care must be taken not to delete any of the `qbuf`s in the linked-list prior to the final use of the presentation element. Note however, than the presentation element may be de-allocated at anytime without affecting the `qbuf`s in the linked-list.

The routine `prim2qb` performs the inverse operation:

```
struct qbuf *prim2qb (pe)
PE      pe;
```

When examining the contents of a qbuf, it is most efficient to examine each qbuf in the linked list, as in:

```
register struct qbuf *qp;

for (qp = qb -> qb_forw; qp != qb; qp = qp -> qb_forw)
    /* examine qp -> qb_data, qp -> qb_len here */ ;
```

However, some applications may wish to simply convert the qbuf into one string and examine it without worrying about traversing the linked list. The routine qb2str performs this function:

```
char   *qb2str (qb)
struct qbuf *qb;
```

The routine str2qb performs the inverse operation, converting a string to a qbuf.

```
struct qbuf *str2qb (s, len, head)
char   *s;
int     len,
head;
```

The head parameter indicates whether str2qb should allocate a qbuf containing both the head of the linked list and one element (head is non-zero), or just the element itself (head is zero).

The routine qb_pullup can be used to compact a linked list of qbufs into a list containing the qbuf head and one element:

```
int     qb_pullup (qb)
struct qbuf *qb;
```

This routine returns the manifest constant NOTOK on failure, and OK otherwise.

The routine qbuf2pe is used to set-up a presentation stream which reads from a linked list of qbufs and returns the presentation element encoded therein:

```
PE      qbuf2pe (qb, len, result)
struct qbuf *qb;
int      len,
        *result;
```

This routine returns a presentation element on success, and `NULLPE` on failure. In the latter case, the parameter `result` is updated to reflect the presentation stream error.

Finally, the routine `qb_free` will deallocate a linked list of `qbuf`s:

```
void    qb_free (qb)
```

5.4.5 Bitvector

A *bitvector* is an arbitrarily long string of bits (starting with bit `0`) with three operations:

`bit_on`: which turns the indicated bit on;

```
int     bit_on (pe, bitno)
PE      pe;
int     bitno;
```

`bit_off`: which turns the indicated bit off;

```
int     bit_off (pe, bitno)
PE      pe;
int     bitno;
```

and,

`bit_test`: which returns a boolean value telling if the indicated bit was on.

```
int     bit_test (pe, bitno)
PE      pe;
int     bitno;
```

The routine `prim2bit` takes a pointer to a presentation element and builds such an abstraction containing the value encoded therein.

```
PE      prim2bit (pe)
PE      pe;
```

The routine `bit2prim` performs the inverse operation.

```
PE      bit2prim (pe)
PE      pe;
```

There are also some support routines useful mainly with *pepy*. The routine `strb2bitstr` takes a pointer to a character string and an integer containing the number of bits and constructs a bit vector as described above containing the bits. You need to call `bit2prim` to convert it into a presentation element.

```
PE      strb2bitstr (cp, length, class, id)
char   *cp;
int     length;
PElementClass class;
PElementID id;
```

The inverse operation is handled by the function `bitstr2strb`. This takes bit vector containing a bit string and produces a new character string with the appropriate bits set. The second parameter will contain the number of valid bits on return.

```
char   *bitstr2strb (pe, length)
PE      pe;
int    *length;
```

Finally, the routines `int2strb` and `strb2int` are provided to convert an integer containing a bit list into a string of bits suitable for input to `strb2bitstr`, and to perform the inverse operation

```
char   *int2strb (n, length)
int     n;
int     length;

int     strb2int (cp, length)
char   *cp;
int     length;
```

5.4.6 Object Identifier

An *object identifier* represents an ordered set of authoritative designations used for identification. Internally, the `OIDentifier` is used to represent this notion:

```
typedef struct OIDentifier {
    int     oid_nelem;

    unsigned int *oid_elements;
}       OIDentifier, *OID;
#define NULLOID ((OID) 0)
```

This structure contains two elements:

`oid_elements/oid_nelem`: the (ordered) list of sub-identifiers (and the number of elements in the list).

Two object identifiers can be compared with `oid_cmp`, which takes two pointers to `OIDentifier` structures as arguments, and returns `0` if the two structures are identical, `1` otherwise.

```
int     oid_cmp (p, q)
OID     p,
        q;
```

Further, an object identifier can be duplicated with `oid_cpy`. This routine takes a pointer to an `OIDentifier` structure as an argument, and returns a pointer to a new `OIDentifier` structure, or `NULLOID` on error.

```
OID     oid_cpy (oid)
OID     oid;
```

Simlarly, the routine `oid_free` can be used to free an object identifier which has been allocated.

```
void    oid_free (oid)
OID     oid;
```

The routine `sprintoid` can be used to return a null-terminated string describing the object identifier.

```
char   *sprintoid (oid)
OID     oid;
```

The default routine `oid2ode` is identical to the `sprintoid`, but is intended for user customization. For example, whilst `sprintoid` should always be used when an object identifier should be expressed in "dot notation", the user may define a new `oid2ode` which returns symbolic names, if so desired.

The routine `str2oid` can be used as the inverse of `sprintoid`:

```
OID     str2oid (s)
char   *s;
```

Finally, the routine `ode2oid` can be used to fetch an object identifier from the *isobjects*(5) database described in Chapter 9.

```
OID     ode2oid (descriptor)
char   *descriptor;
```

This routine performs a lookup using the `getisobjectbyname` routine and then returns a pointer to a static `OIDentifier` structure containing the desired information. The routine returns `NULLOID` on failure.

The routines `prim2oid` and `oid2prim` are used to translate between a machine-specific internal form and the machine-independent form.

```
OID     prim2oid (pe)
PE      pe;

PE      oid2prim (o)
OID     o;
```

In actuality, `oid2prim` is really a macro which calls the routine `obj2prim` which builds a presentation element containing the object identifier given and sets the `pe_class` an `pe_id` fields of the presentation element to the desired values.

```
PE      obj2prim (o, class, id)
OID     o;
PElementClass class;
PElementID id;
```

5.4.7 Timestring

A *timestring* represents a date/time in many forms. Currently, two forms are supported: *universal time* and *generalized time*. Internally, the `UTCtime` structure is used to represent both notions:

```
typedef struct UTCtime {
    int     ut_year;
    int     ut_mon;
    int     ut_mday;
    int     ut_hour;
    int     ut_min;
    int     ut_sec;

    int     ut_msec;

    int     ut_zone;

    int     ut_flags;
}       UTCtime, *UTC;
#define NULLUTC ((UTC) 0)
```

This structure contains several elements:

`ut_year`: the year, either since the current century (e.g., `86`), or absolute (e.g., `1986`);

`ut_mon`: the month of the year, in the range `1..12`;

`ut_mday`: the day of the month, in the range `1..31`;

`ut_hour`: the hour of the day, in the range `0..23`;

`ut_min`: the minute of the hour, in the range `0..59`;

`ut_sec`: (optionally) the second of the minute, in the range `0..59`;

`ut_usec`: (optionally) fractions of a second, in microseconds;

`ut_zone`: (optionally) the timezone, expressed as minutes from UT; and,

`ut_flags`: various flags describing the timestring:

UT_ZONE: the ut_zone element is present;

UT_SEC: the ut_sec element is present; and,

UT_USEC: the ut_usec element is present.

The routine prim2utct takes a pointer to a presentation element and returns the universal time encoded therein.

```
UTC     prim2utct (pe)
PE      pe;
```

The routine utct2prim is a macro which performs the inverse operatoin. It behaves as if it was defined as:

```
PE      utct2prim (u)
UTC     u;
```

The routine prim2gent takes a pointer to a presentation element and returns the universal time encoded therein.

```
UTC     prim2gent (pe)
PE      pe;
```

The routine gent2prim is a macro which performs the inverse operatoin. It behaves as if it was defined as:

```
PE      gent2prim (u)
UTC     u;
```

In actuality, both utct2prim and gent2prim call the routine time2prim which builds a presentation element containing the universal or general time given and sets the pe_class an pe_id fields of the presentation element to the desired values.

```
PE      time2prim (ut, generalized, class, id)
UTC     u;
int generalized;
PElementClass class;
PElementID id;
```

To get a null-terminated string representation of a UTCtime structure, the macros utct2str and gent2str can be used. Each take a single argument, a pointer to a UTCtime structure. These call the routine time2str:

```
char   *time2str (ut, generalized)
UTC     u;
int     generalized;
```

Both call the routine `prim2time`:

```
UTC     prim2time (pe, generalized)
PE      pe;
int     generalized;
```

The routines `str2utct` and `str2gent` perform the inverse operations:

```
UTC     str2utct (cp, len)
char   *cp;
int     len;

UTC     str2gent (cp, len)
char   *cp;
int     len;
```

Each take a character-pointer and a length-indicator as arguments and return a `UTCtime` structure on success. Otherwise, the manifest constant `NULLUTC` is returned.

There are also some utility routines to aid in converting between `UTCtime` and UNIX `tm` time structures. The routine `gtime` is the inverse of *gmtime*(3), it takes a time structure and returns a long-valued number.

```
long    gtime (tm)
struct tm *tm;
```

The routine `ut2tm` returns a pointer to a static time structure which has been initialized by its `UTC` argument.

```
struct tm *ut2tm (ut)
UTC     ut;
```

Finally, the routine `tm2ut` performs the inverse operation, initializing a `UTC` argument by using a time structure argument.

```
void     tm2ut (tm, ut)
struct tm *tm;
UTC     ut;
```

5.4.8 Sets and Sequences

Two list disciplines are implemented: *sets*, in which each member is distinguished by a unique identifier; and, *sequences*, in which each element is distinguished by its offset from the head of the list. It should be noted that these definitions of sets and sequences do not exactly match those defined in ASN.1. In particular, the ASN.1 structure `SET␣OF` must be defined as a type `SET`, but be built up with the `seq_add` construct, as these elements all have the same type. On both types of lists, the macro `first_member` returns the first member in the list, while `next_member` returns the next member. These macros behave as if they were defined as:

```
PE      first_member (head)
PE      head;

PE      next_member (head, member)
PE      head,
        member;
```

There are three operations on sets:

`set_add`: adds a new member to the set;

```
int     set_add (pe, r)
PE      pe,
        r;
```

The routine `set_addon` is an efficient version of `set_add` that is used when adding several consecutive members to a set.

`set_del`: removes the identified member from the set;

```
int     set_del (pe, class, id)
PE      pe;
PElementClass class;
PElementID id;
```

and,

`set_find`: locates the identified member.

```
PE      set_find (pe, class, id)
PE      pe;
PElementClass class;
PElementID id;
```

The routines `prim2set` and `set2prim` are used to translate between presentation elements and sets.

```
PE      prim2set (pe)
PE      pe;

PE      set2prim (pe)
PE      pe;
```

In Figure 5.1, a convenient way of stepping through all the members of a set is presented.

There are three operations on sequences:

`seq_add`: adds a new member `r` to the sequence `pe`, at the given offset, an offset of `-1` will add the element to the end of the sequence;

```
int     seq_add (pe, r, offset)
PE      pe,
        r;
int     offset;
```

The routine `seq_addon` is an efficient version of `seq_add` that is used when adding several consecutive elements to a set.

`seq_del`: removes the identified member from the sequence;

```
int     seq_del (pe, offset)
PE      pe;
int     offset;
```

and,

`seq_find`: locates the identified element.

```
PE      seq_find (pe, offset)
PE      pe;
int     offset;
```

```
#include <isode/psap.h>

...

register struct PE pe, p;

...

for (p = first_member (pe); p; p = next_member (pe, p))
    switch (PE_ID (p -> pe_class, p -> pe_id)) {
            case PE_ID (PE_CLASS_UNIV, PE_DEFN_IA5S):
                ...

            case PE_ID (PE_CLASS_UNIV, PE_DEFN_T61S):
                ...

            ...
    }

...
```

Figure 5.1: Stepping through a Sequence

```
#include <isode/psap.h>

...

register struct PE pe, p;

...

for (p = first_member (pe); p; p = next_member (pe, p))
   switch (p -> pe_offset) {
         case 0:
            ...

         case 1:
            ...

         ...
   }

...
```

Figure 5.2: Stepping through a Set

The routines `prim2seq` and `seq2prim` are used to translate between presentation elements and sequences.

```
PE      prim2seq (pe)
PE      pe;

PE      seq2prim (pe)
PE      pe;
```

In Figure 5.2, a convenient way of stepping through all the members of a sequence is presented.

5.5 Inline CONStructors

Please read the following important message.

NOTE:	The facilities described in this section are to be used only as a last resort. They are provided for compatibility with less-rich systems.

When interfacing external software on top of various libraries, such as *librosap*(3n), the Remote Operations library, arguments which are expected to be `PElement`s may be available instead as the raw encoded octets. It is inefficent to convert this into a `PElement`, pass the structure to the library, which then just encodes it again. To avoid this behavior, the routine, `str2pe`, may be used:

```
PE      str2pe (s, len, advance, result)
char   *s;
int     len,
       *advance,
       *result;
```

The parameters to this procedure are:

`s/len`: the encoded string (and its length);

`advance`: a pointer to an integer-valued location indicating how far to advance the string to find the next ASN.1 object (use the manifest constant `NULLIP` if you aren't interested in this), and,

`result`: a pointer to an integer-valued location which indicates, if the call fails, the reason for the failure.

On success, `str2pe` returns a `PElement` called an *Inline CONStructor*. This is a special kind of `PElement` which really just contains a pointer to the original string. The underyling libraries now how to manipulate Inline CONStructors appropriately, e.g., when building an SSDU. On failure, `str2pe` returns the manifest constant `NULLPE` and the `result` parameter contains an error code listed in Table Table 5.1 on page 116.

For the inverse operation, the appropriate library must be informed that the ASN.1 objects associated with incoming indications should be returned in Inline CONStructor form. This mechanism is still being refined, and is not currently documented.

However, the routine used by the libraries in order to unwrap an encoded string is available:

```
PE      qb2pe (qb, len, depth, result)
struct qbuf *qb
int     len,
        depth,
       *result;
```

The parameters to this procedure are:

`qb/len`: a doubly-linked list of `qbuf`s containing, e.g., an SSDU, and the total length of the data in the `qbuf`s;

`depth`: the level of unwrapping to permit; and,

`result`: a pointer to an integer-valued location which indicates, if the call fails, the reason for the failure.

On success, `qb2pe` returns a `PElement` which may ultimately contain Inline CONStructors. On failure, `qb2pe` returns the manifest constant `NULLPE` and the `result` parameter contains an error code listed in Table Table 5.1.

5.6 Compiling and Loading

Programs which manipulate presentation streams or presentation elements should include the header file `<isode/psap.h>`. These programs should also be loaded with `-lpsap` which contains the routines described in this chapter.

5.7 An Example

Let's consider how one might read a presentation stream from the standard input. The process is simple: we create a new presentation stream and then start the loop which reads from the stream until it is exhausted. This is shown in Figure 5.3. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

```
#include <stdio.h>
#include <isode/ppkt.h>

/* ARGSUSED */

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

    if ((ps = ps_alloc (std_open)) == NULLPS)
        error ("ps_alloc");
    if (std_setup (ps, stdin) == NOTOK)
        error ("std_setup: %s", ps_errno (ps -> ps_errno));

    for (;;) {
        if ((pe = ps2pe (ps)) == NULLPE)
            if (ps -> ps_errno)
                error ("ps2pe: %s", ps_errno (ps -> ps_errno));
            else
                break;          /* end-of-file */

/* process "pe" here... */

            pe_free (pe);
    }

    ps_free (ps);

    exit (0);
}
```

Figure 5.3: Stepping through a Presentation Stream

5.8 For Further Reading

The language for the Abstract Syntax Notation One (ASN.1) is specified in [ISO87a] and [CCI88e], while the encoding rules are defined in [ISO87b] and [CCI88e]. The older 1984 CCITT recommendations, containing both definitions, is [CCI84a], which is often referred to as X.409. The ASN.1 is a superset of the X.409.

Part IV

Databases

Chapter 6

The ISO Aliases Database

The database *isoaliases* in the ISODE `ETCDIR` directory (usually */usr/etc/*) contains a simple mapping between names (terse strings) and values (e.g., user-friendly names and distinguished names).

The database itself is an ordinary ASCII text file containing an entry for each locally defined alias. Each entry contains

- the alias, a simple string; and,
- a user-friendly name or a distinguished name.

Blanks and/or tab characters are used to seperate items. However, double-quotes may be used to prevent separation for items containing embedded whitespace. The sharp character ('`#`') at the beginning of a line indicates a commentary line.

6.1 Accessing the Database

The *libacsap*(3n) library contains the routines used to access the database. There is one high-level routine, `alias2name` which returns the value which corresponds to an alias in the database.

```
char   *alias2name (name)
char   *name;
```

The parameter to this procedure is:

`name`: the alias to lookup.

This returns the manifest constant `NULLCP` if the given alias is not in the database.

In order to load specific aliases other than those read in the *isoaliases*(5) file, use the routine `add_alias`:

```
int     add_alias (name, value)
char   *name,
       *value;
```

The parameters to this procedure are:

`name`: the alias to enter; and,

`value`: its value.

This returns the manifest constant `NOTOK` if the given alias cannot be added.

6.2 User-Specific Aliases

By default a user-specific aliases database is consulted before the system-wide aliases file. The user-specific file is called *$HOME/.isode_aliases* in the user's home directory.

Chapter 7

The ISODE Entities Database

The database *isoentities* in the ISODE `ETCDIR` directory (usually */usr/etc/*) contains a simple mapping between application-entity information and presentation addresses. This database is used by the stub-directory service.

NOTE:	Use of this database is deprecated. Consult Chapter 10 on page 160 for further information.

The database itself is an ordinary ASCII text file containing information regarding the known application-entities on the network. Each entry contains:

- the object descriptor of the application-entity information, realized as a designator-qualifier (host/service) pair (the distinguished designator `default` is used for a template entry);
- the object identifier of the application-entity information expressed in dot-notation (if no application-entity information is desired, the string "NULL" should be used instead); and,
- the presentation address expressed in the string format described in Section 2.2.1 on page 20 in *Volume One*.

  Note that since double-quotes are often used in the new string format, it is **very** important to quote them correctly in the *isoentities*(5) file. Usually preceeding the first character of the address with backslash ("\") is adequate.

Blanks and/or tab characters are used to seperate items. However, double-quotes may be used to prevent separation for items containing embedded whitspace. This is useful for defining the presetnation address. The sharp character ('#') at the beginning of a line indicates a commentary line.

It is suggested for readability purposes that a blank line should seperate entries.

RFC1085 Support

Since applications using RFC1085 (the lightweight presentation protocol) usually demultiplex on the basis of TCP or UDP port, a further definition for the qualifier is placed in */etc/services*, one of:

```
qualifier    portno/lpp
qualifier    portno/tcp
qualifier    portno/udp
```

The first alternative says that the service lives on both TCP and UDP; the second alternative says that the service lives on TCP only; and, the third alternative says that the service lives on UDP only.

7.1 Accessing the Database

The *libacsap*(3n) library contains the routines used to access the database. The high-level routines, `_str2aei`, which returns application-entity information, and `aei2addr`, which maps the application-entity information into a presentation address have already been discussed in Section 2.2.1. Similarly, the "old-style" routines, `is2paddr`, `is2saddr`, `is2taddr`, which maps a host and service directly into a prsentation, session, or transport address (respectively) are discussed in Sections 2.2, 3.2, and 4.1 of *Volume Two* (respectively). Hence, for the remainder of this chapter, we need only consider the internal form used by the library in querying the database.

The `isoentity` structure is the internal form.

```
struct isoentity {
    OIDentifier ie_identifier;
    char    *ie_descriptor;
```

```
        struct PSAPaddr ie_addr;
    };
```

The elements of this structure are:

`ie_identifier`: an object identifier used to form application-entity information;

`ie_identifier`: the corresponding object descriptor; and,

`ie_addr`: the presentation address.

The routine `getisoentity` reads the next entry in the database, opening the database if necessary.

```
    struct isoentity *getisoentity ()
```

It returns the manifest constant `NULL` on error or end-of-file.

The routine `setisoentity` opens and rewinds the database.

```
    int     setisoentity (f)
    int     f;
```

The parameter to this procedure is:

`f`: the "stayopen" indicator, if non-zero, then the database will remain open over subsequent calls to the library.

The routine `endisoentity` closes the database.

```
    int     endisoentity ()
```

Both of these routines return non-zero on success and zero otherwise.

Chapter 8

The ISO Macros Database

The database *isomacros* in the ISODE `ETCDIR` directory (usually */usr/etc/*) contains a simple mapping between user-friendly strings and network addresses. This database is used by when trying to resolve textual respresentations of network addresses (as desccribed in Section 2.2.1) for use with the network.

The database itself is an ordinary ASCII text file containing an entry for each locally defined macro. Each entry contains

- the macro, a simple string; and,
- the prefix of the corresponding network address.

Blanks and/or tab characters are used to seperate items. However, double-quotes may be used to prevent separation for items containing embedded whitspace. The sharp character ('`#`') at the beginning of a line indicates a commentary line.

Unlike the other databases in the ISODE, the user may not directly access this file. The routine `str2paddr` and `paddr2str` use this automatically.

8.1 User-Specific Macros

By default a user-specific macros database is consulted before the system-wide macros file. The user-specific file is called *$HOME/.isode_macros* in the user's home directory.

Chapter 9

The ISODE Objects Database

The database *isobjects* in the ISODE `ETCDIR` directory (usually */usr/etc/*) contains a simple mapping between object descriptors and object identifiers.

The database itself is an ordinary ASCII text file containing information regarding the known objects on the host. Each line contains

- the descriptor of the object, a simple string; and,
- the corresponding object identifier.

Blanks and/or tab characters are used to separate items. However, double-quotes may be used to prevent separation for items containing embedded whitespace. The sharp character ('`#`') at the beginning of a line indicates a commentary line.

9.1 Accessing the Database

The *libpsap*(3) library contains the routines used to access the database. These routines ultimately manipulate an `isobject` structure, which is the internal form.

```
struct isobject {
    char          *io_descriptor;

    OIDentifier    io_identity;
};
```

The elements of this structure are:

> `io_descriptor`: the object descriptor; and,
>
> `io_identity`: the object identifier.

The routine `getisobject` reads the next entry in the database, opening the database if necessary.

```
struct isobject *getisobject ()
```

It returns the manifest constant `NULL` on error or end-of-file.

The routine `setisobject` opens and rewinds the database.

```
int     setisobject (f)
int     f;
```

The parameter to this procedure is:

> `f`: the "stayopen" indicator, if non-zero, then the database will remain open over subsequent calls to the library.

The routine `endisobject` closes the database.

```
int     endisobject ()
```

Both of these routines return non-zero on success and zero otherwise.

There are two routines used to fetch a particular entry in the database. The routine `getisobjectbyname` maps object descriptors into the internal form.

```
struct isobject *getisobjectbyname (descriptor)
char   *descriptor;
```

The parameter to this procedure is:

> `descriptor`: the descriptor of the object.

and returns the `isobject` structure describing that object. On failure, the manifest constant `NULL` is returned instead.

The routine `getisobjectbyoid` performs the inverse function.

```
struct isobject *getisobjectbyoid (oid)
OID     oid;
```

The parameter to this procedure is:

`oid`: the identifier of the object.

On a successful return, an `isobject` structure describing the object is returned.

Chapter 10

Defining New Services

The OSI Directory is used to register new services. The steps involved are simple:

- the application context and abstract syntax for the service is registered in the *isobjects*(5) file, this allows for your program to use the textual designator for these values rather than the object identifier form; and,
- an entry is created in the Directory containing information indicating where the service resides in the network, and, optionally, what UNIX program will be invoked whenever there is an incoming connection for the service.

Please read the following important message.

NOTE:	Consult the *READ-ME* file for information on how to generate the initial Directory entries for your system.

10.1 Standard Services

The "standard" approach is used when an incoming connection should result in *tsapd* invoking the service.

First, edit the *isoentities*(5) database (described in Chapter 7), and add the appropriate lines to define

- the abstract syntax used by the service (the description of the data structures exchanged at the application layer); and,

- the application context providing the service (the description of the service elements contained in the application layer along with the interactions between them and the presentation layer).

These are object identifiers (as described in Section 5.4.6 on page 135). The object identifier tree `1.17.2` has been usurped for defining local services using *The ISO Development Environment*. Choose `n`, where `n` is the lowest unassigned number in the `1.17.2.n` subtree, for use by the service (`n` should be an integer greater than `0`). Now edit the *isobjects*(5) file thusly:

```
"local service pci"    1.17.2.n.1
"local service"        1.17.2.n.2
```

Second, add an entry to the Directory so that initiator entities can locate the service. The following attributes will have to be entered:

commonName: The name that the entry is registered under, (e.g., `cn=servicestore`).

presentationAddress: The location in the network where the service resides, consisting of a transport-selector and one or more network addresses (if your host is multi-homed). It is easiest to use a 16–bit binary space for the transport selector: choose `t`, where `t` is the lowest unassigned TSAP ID between `1024` and `2047` inclusive.

supportedApplicationContext: The application context providing the service. Use the same string that you entered in the *isobjects*(5) file.

execVector: The command string which *tsapd* will invoke for each incoming connection. The first token is interpreted relative to the ISODE `SBINDIR` directory (usually */usr/etc/*).

So, the entry looks something like this:

```
objectClass= top & applicationEntity & quipuObject
objectClass= iSODEApplicationEntity
cn= servicestore
presentationAddress= #t/Internet=mydomainame
supportedApplicationContext= local service
acl=
execVector= program arg1 arg2 ... argN
```

10.2 Static Servers

In Section 2.5, two different server disciplines, the dynamic and the static approaches, were described. Thus far, this chapter has described the dynamic approach. The distinguishing mechanism of this discipline depends on whether the `objectClass` attribute for the entry contains the value

```
iSODEApplicationEntity
```

If so, then the entry describes a dynamic approach, and *tsapd* will listen for incoming connections.

If not, then the entry does not contain a `execVector` attribute, and *tsapd* will ignore the entry in the Directory. In order to avoid listening conflicts with *tsapd*, the network address chosen should be different:

- If the network-address is TCP-based, use a TCP port, `p`, that is different than the one used by *tsapd* (which, by default, is 102), e.g.,

  ```
  presentationAddress= #t/Internet=mydomainame+p
  ```

- If the network-address is X.25–based, use an X.25 protocol-ID, `p`, that is different than the one used by *tsapd* (which, by default is empty), e.g.,

  ```
  presentationAddress= #t/Int-X25=mydomainame+PID+p
  ```

- If the network-address is CLNS–based, then simply choose any unused transport-selector, `p`, e.g.,

  ```
  presentationAddress= #t/NS+470005...
  ```

After making a suitable entry in the Directory, the server program must be started (either by hand or automatically from some system startup file, e.g., */etc/rc.local*) without any arguments. The server program, using the `isodeserver` routine described on page 44 will then perform the appropriate actions to start listening on the desired addresses.

Finally, note that in order for the `isodeserver` routine to work correctly, you will need an entry in the *isoaliases*(5) file mapping the local host name (as returned by `PLocalHostName`) into a Directory Distinguished Name, e.g.,

```
hubris    hubris, cs, university college london, gb
```

This is necessary so that the call to `_str2aei` (or `str2aeinfo`) can find the appropriate application-entity information which is subsequently passed to `isodeserver`.

Part V

Appendices

Appendix A

Old-Style Associations

As discussed in Section 2.1, there are several ways to establish an association. In the interests of compatibility with previously specified applications, the *librosap*(3n) and *librtsap*(3n) libraries support "old-style" mechanisms for binding associations. These are discussed in this appendix. Use of these facilities for new applications is strongly discouraged.

A.1 Remote Operations

Under old-style association handling for remote operations, the mechanism used for establishing the association determines the remote operations service discipline to be used. If the basic service discipline is desired, then continue reading this section which describes the addressing and initialization mechanisms for native ROS associations. These make use of the RO-BEGIN and RO-END primitives. Instead, if the advanced service discipline is desired, then read Section A.2 on page 175 which describes how associations are established and released by the reliable transfer service (RTS) using the X.410 OPEN and X.410 CLOSE primitives. (Be sure to also read Section 3.3.1 on page 62).

A.1.1 Addresses

Addresses at the remote operations service access point are represented by the `RoSAPaddr` structure.

```
struct RoSAPaddr {
    struct SSAPaddr roa_addr;

    u_short     roa_port;
};
```

This structure contains two elements:

`roa_addr`: the session address, as described in Section 3.2 on page 42 of *Volume Two*; and,

`roa_port`: the port number of the entity residing above the RoSAP.

In Figure A.1, an example of how one constructs the RoSAP address for the directory services entity on host `RemoteHost` is presented. (The routine `is2saddr` is described on page 42 of *Volume Two*.)

A.1.2 Association Establishment

The *librosap*(3n) library distinguishes between the user which started an association, the *initiator*, and the user which subsequently was bound to the association, the *responder*. We sometimes term these two entities the *client* and the *server*, respectively.

Server Initialization

The *tsapd*(8c) daemon, upon accepting a connection from an initiating host, consults the ISO services database to determine which program on the local system implements the desired RoSAP entity. In the case of the remote operations service, the *tsapd* program contains the bootstrap for the remote operations provider. The daemon will again consult the ISO services database to determine which program on the system implements the desired RoSAP entity.

Once the program has been ascertained, the daemon runs the program with any arguments listed in the database. In addition, it appends some *magic arguments* to the argument vector. Hence, the very first action performed by the responder is to re-capture the RoSAP state contained in the magic arguments. This is done by calling the routine `RoInit`, which on a successful return, is equivalent to a RO-BEGIN.INDICATION from the remote operations service provider.

```
#include <isode/rosap.h>
#include <isode/isoservent.h>

...

register struct SSAPaddr *sa;
struct RoSAPaddr roas;
register struct RoSAPaddr *roa = &roas;
register struct isoservent *is;

...

if ((is = getisoserventbyname ("directory", "rosap")) == NULL)
    error ("rosap/directory");

/* RemoteHost is the host we're interested in,
   e.g., "gremlin.nrtc.northrop.com" */

roa -> roa_port = is -> is_port;
if ((is = getisoserventbyname ("ros", "ssap")) == NULL)
    error ("ssap/ros");
if ((sa = is2saddr (RemoteHost, NULLCP, (struct isoservent *) is))
            == NULLSA)
    error ("address translation failed");
roa -> roa_addr = *sa;

...
```

Figure A.1: Constructing the RoSAP address for the Directory Services entity

```
int     RoInit (vecp, vec, ros, roi)
int     vecp;
char  **vec;
struct RoSAPstart *ros;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

`ros`: a pointer to a `RoSAPstart` structure, which is updated only if the call succeeds; and,

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

If `RoInit` is successful, it returns information in the `ros` parameter, which is a pointer to a `RoSAPstart` structure.

```
struct RoSAPstart {
    int     ros_sd;

    struct RoSAPaddr ros_initiator;

    u_short ros_port;

    PE      ros_data;
};
```

The elements of this structure are:

`ros_sd`: the association-descriptor to be used to reference this association;

`ros_initiator`: the "unique identifier" of the initiator (containing the initiator's host address);

`ros_port`: the application number the initiator wishes to use to govern the association; and `ros_data`: any initial data (this is a pointer to a `PElement` structure, which is fully explained in Chapter 5).

Note that the `ros_data` element is allocated via *malloc*(3) and should be released using the `ROSFREE` macro when no longer referenced. The `ROSFREE` macro behaves as if it was defined as:

```
void    ROSFREE (ros)
struct RoSAPstart *ros;
```

The macro frees only the data allocated by `RoInit`, and not the `RoSAPstart` structure itself. Further, `ROSFREE` should be called only if the call to the `RoInit` routine returned `OK`.

If the call to `RoInit` is not successful, then a RO-P-REJECT.INDICATION event is simulated, and the relevant information is returned in an encoded `RoSAPindication` structure.

After examining the information returned by `RoInit` on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the association. For either response, the responder should use the `RoBeginResponse` routine (which corresponds to the RO-BEGIN.RESPONSE action).

```
int     RoBeginResponse (sd, status, data, roi)
int     sd;
int     status;
PE      data;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`status`: the acceptance indicator (either `ROS_ACCEPT` if the association is to be accepted, or one of the "fatal" user-initiated rejection codes listed in Table 3.1 on page 60);

`data`: any initial data if the association is to be accepted; and,

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

If the call to RoBeginResponse is successful, and the status parameter is set to ROS_ACCEPT, then association establishment has now been completed. If the call is successful, but the status parameter is not ROS_ACCEPT, then the association has been rejected and the responder may exit. Otherwise, if the call fails and the reason is "fatal", then the association is closed.

Client Initialization

A program wishing to associate itself with another user of remote operation services calls the RoBeginRequest routine, which corresponds to the RO-BEGIN.REQUEST action.

```
int     RoBeginRequest (called, data, roc, roi)
struct  RoSAPaddr *called;
PE      data;
struct RoSAPconnect *roc;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

called: the RoSAP address of the responder;

data: any initial data;

roc: a pointer to a RoSAPconnect structure, which is updated only if the call succeeds; and,

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

If the call to RoBeginRequest is successful (this corresponds to a RO-BEGIN.CONFIRMATION event), it returns information in the roc parameter, which is a pointer to a RoSAPconnect structure.

```
struct RoSAPconnect {
    int     roc_sd;

    int     roc_result;

    PE      roc_data;
};
```

The elements of this structure are:

`roc_sd`: the association-descriptor to be used to reference this association;

`roc_result`: the association response; and,

`roc_data`: any initial data if the association was accepted.

If the call to `RoBeginRequest` is successful, and the `roc_result` element is set to `ROS_ACCEPT`, then association establishment has completed. If the call is successful, but the `roc_result` element is not `ROS_ACCEPT`, the the association attempt has been rejected, consult Table 3.1 to determine the reason for the reject. Otherwise, if the call fails then the association is not established and the `RoSAPpreject` structure of the `RoSAPindication` discriminated union has been updated.

Note that the `roc_data` element is allocated via *malloc*(3) and should be released using the `ROCFREE` macro when no longer referenced. The `ROCFREE` macro behaves as if it was defined as:

```
void    ROCFREE (roc)
struct RoSAPconnect *roc;
```

The macro frees only the data allocated by `RoBeginRequest`, and not the `RoSAPconnect` structure itself. Further, `ROCFREE` should be called only if the call to the `RoBeginRequest` routine returned `OK`.

Note that in the basic service the arguement parameter to an Invoke request cannot be NULL. As well only the initiator may make invoke requests.

A.1.3 Association Release

The `RoEndRequest` routine is used to request the release of an association, and corresponds to a RO-END.REQUEST action. This action may be taken by only the initiator of an association.

```
int     RoEndRequest (sd, priority, roi)
int     sd;
int     priority;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`priority`: the priority of this request; and,

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

If the call to `RoEndRequest` is successful, then the association has been released. Otherwise if the call fails and the error is not fatal, then the association remains established and the `RoSAPpreject` structure contained in the `RoSAPindication` parameter `roi` contains the reason for the failure.

Upon receiving a RO-END.INDICATION event, the user is required to generate a RO-END.RESPONSE action using the `RoEndResponse` routine.

```
int     RoEndResponse (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.

If the call to `RoEndResponse` is successful, then the association has been released.

A.1.4 An Example

Let's consider how one might construct a generic server which uses remote operations services. This entity will use an asynchronous interface and the basic service discipline. Note that we could have selected the advanced service discipline by using `RtBInit` instead of `RoInit`, `RtBeginResponse` instead of `RoBeginResponse`, and `RtEndResponse` instead of `RoEndResponse` (respectively). This is demonstrated later in this appendix.

There are two parts to the program: initialization and the request/reply loop. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure A.2, the initialization steps for the generic responder, including the outer *C* wrapper, is shown. First, the RoSAP state is re-captured by calling `RoInit`. If this succeeds, then the association is authenticated and any command line arguments are parsed. Assuming that the responder is satisfied with the proposed association, it calls `RoBeginResponse` to accept the association. Then, `RoSetIndications` is set to specify an asynchronous event handler. Finally, the main request/reply loop is realized. The server simply uses *pause*(2) to wait for the next event.

Figure 3.2 on page 77 contains the definition of the `ros_indication` routine and associated routines. Since the remote operations service was used to establish the association, a different closing handler must be used. This is shown in Figure A.3.

A.2 Reliable Transfer

Under old-style association handling for reliable transfer, the X.410 OPEN and X.410 CLOSE primitives are used.

A.2.1 Addresses

Addresses at the reliable transfer service access point are represented by the `RtSAPaddr` structure.

```
struct RtSAPaddr {
    struct SSAPaddr rta_addr;

    u_short     rta_port;
};
```

This structure contains two elements:

`rta_addr`: the session address, as described in Section 3.2 on page 42 of *Volume Two*; and,

`rta_port`: the port number of the entity residing above the RtSAP.

In Figure A.4, an example of how one constructs the RtSAP address for the message transfer entity on host `RemoteHost` is presented. (The routine `is2saddr` is described on page 42 of *Volume Two*.)

```
#include <stdio.h>
#include "generic.h"            /* defines OPERATIONS and ERRORS */
#include <isode/rosap.h>
#include <isode/isoservent.h>


int     ros_indications ();


main (argc, argv, envp)
int     argc;
char  **argv,
      **envp;
{
    int     result,
            sd;
    struct RoSAPstart   ross;
    register struct RoSAPstart *ros = &ross;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (RoInit (argc, argv, ros, roi) == NOTOK)
        error ("initialization fails:  %s", RoErrString (rop -> rop_reason));

    sd = ros -> ros_sd;
/*  do authentication using ros -> ros_data here... */
    ROSFREE (ros);

/* read command line arguments here... */

/* could use ROS_VALIDATE or ROS_BUSY instead and then exit */

    if (RoBeginResponse (sd, ROS_ACCEPT, NULLPE, roi) == NOTOK)
        error ("RO-BEGIN.RESPONSE: %s", RoErrString (rop -> rop_reason));

    if (RoSetIndications (sd, ros_indication, roi) == NOTOK)
        error ("RoSetIndications:  %s", RoErrString (rop -> rop_reason));

    for (;;)
        pause ();
}
```

Figure A.2: Initializing the generic ROS responder

```
...

static int  ros_end (sd, roe)
int     sd;
struct RoSAPend *roe;
{
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (RoEndResponse (sd, roi) == NOTOK)
        error ("RO-END.RESPONSE: %s", RoErrString (rop -> rop_reason));

    exit (0);
}
```

Figure A.3: Finalizing the generic ROS responder

A.2.2 Association Establishment

The *librtsap*(3n) library distinguishes between the user which started an association, the *initiator*, and the user which was subsequently bound to the association, the *responder*. We sometimes term these two entities the *client* and the *server*, respectively.

Server Initialization

The *tsapd*(8c) daemon, upon accepting a connection from an initiating host, consults the ISO services database to determine which program on the local system implements the desired SSAP entity. In the case of the reliable transfer service, the *tsapd* program contains the bootstrap for the reliable transfer provider. The daemon will again consult the ISO services database to determine which program on the system implements the desired RtSAP entity.

Once the program has been ascertained, the daemon runs the program with any arguments listed in the database. In addition, it appends some *magic arguments* to the argument vector. Hence, the very first action performed by the responder is to re-capture the RtSAP state contained in the magic arguments. This is done by calling the routine `RtBInit`, which on a successful return, is equivalent to a X.410 OPEN.INDICATION from the reliable transfer service provider.

```
#include <isode/rtsap.h>
#include <isode/isoservent.h>

...

register struct SSAPaddr *sa;
struct RtSAPaddr rtas;
register struct RtSAPaddr *rta = &rtas;
register struct isoservent *is;

...

if ((is = getisoserventbyname ("p1", "rtsap")) == NULL)
    error ("rtsap/p1");

/* RemoteHost is the host we're interested in,
   e.g., "gremlin.nrtc.northrop.com" */

rta -> rta_port = is -> is_port;
if ((is = getisoserventbyname ("rts", "ssap")) == NULL)
    error ("ssap/rts");
if ((sa = is2saddr (RemoteHost, NULLCP, (struct isoservent *) 0))
            == NULLSA)
    error ("address translation failed");
rta -> rta_addr = *sa;

...
```

Figure A.4: Constructing the RtSAP address for the Message Transfer entity

```
int     RtBInit (vecp, vec, rts, rti)
int     vecp;
char  **vec;
struct RtSAPstart *rts;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

`rts`: a pointer to a `RtSAPstart` structure, which is updated only if the call succeeds; and,

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

If `RtBInit` is successful, it returns information in the `rts` parameter, which is a pointer to a `RtSAPstart` structure.

```
struct RtSAPstart {
    int     rts_sd;

    struct RtSAPaddr rts_initiator;

    int     rts_mode;
#define RTS_MONOLOGUE   0
#define RTS_TWA         1

    int     rts_turn;
#define RTS_INITIATOR   0
#define RTS_RESPONDER   1

    u_short rts_port;

    PE      rts_data;
};
```

The elements of this structure are:

`rts_sd`: the association-descriptor to be used to reference this association;

`rts_initiator`: the address of the initiator;

`rts_mode`: the dialogue mode (either monologue or two-way alternate),

`rts_turn`: the owner of the turn initially;

`rts_port`: the application number the initiator wishes to use to govern the association; and

`rts_data`: any initial data (this is a pointer to a `PElement` structure, which is fully explained in Chapter 5).

Note that the `rts_data` element is allocated via *malloc*(3) and should be released by using the `RTSFREE` macro when no longer referenced. The `RTSFREE` macro behaves as if it was defined as:

```
void    RTSFREE (rts)
struct RtSAPstart *rts;
```

The macro frees only the data allocated by `RtBInit`, and not the `RtSAPstart` structure itself. Further, `RTSFREE` should be called only if the call to the `RtBInit` routine returned `OK`.

If the call to `RtBInit` is not successful, then a RT-P-ABORT.INDICATION event is simulated, and the relevant information is returned in an encoded `RtSAPindication` structure.

After examining the information returned by `RtBInit` on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the association. For either response, the responder should use the `RtBeginResponse` routine (which corresponds to the X.410 OPEN.RESPONSE action).

```
int     RtBeginResponse (sd, status, data, rti)
int     sd;
int     status;
PE      data;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`status`: the acceptance indicator (either `RTS_ACCEPT` if the association is to be accepted, or one of the "fatal" user-initiated rejection codes, other tha `RTS_REJECT`, listed in Table 4.1 on page 97);

`data`: any initial data if the association is to be accepted; and,

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

If the call to `RtBeginResponse` is successful, and the `status` parameter is set to `RTS_ACCEPT`, then association establishment has now been completed. If the call is successful, but the `status` parameter is not `RTS_ACCEPT`, then the association has been rejected and the responder may exit. Otherwise, if the call fails and the reason is "fatal", then the association is closed.

Client Initialization

A program wishing to associate itself with another user of reliable transfer services calls the `RtBeginRequest` routine, which corresponds to the X.410 OPEN.REQUEST action.

```
int     RtBeginRequest (called, mode, turn, data, rtc, rti)
struct  RtSAPaddr *called;
int     mode,
        turn;
PE      data;
struct RtSAPconnect *rtc;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`called`: the RtSAP address of the responder;

`mode`: the dialogue mode;

`turn`: who gets the initial turn;

`data`: any initial data;

`rtc`: a pointer to a `RtSAPconnect` structure, which is updated only if the call succeeds; and,

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

If the call to `RtBeginRequest` is successful (this corresponds to a X.410 OPEN.CONFIRMATION event), it returns information in the `rtc` parameter, which is a pointer to a `RtSAPconnect` structure.

```
struct RtSAPconnect {
    int     rtc_sd;

    int     rtc_result;

    PE      rtc_data;
};
```

The elements of this structure are:

`rtc_sd`: the association-descriptor to be used to reference this association;

`rtc_result`: the association response; and,

`rtc_data`: any initial data if the association was accepted.

If the call to `RtBeginRequest` is successful, and the `rtc_result` element is set to `RTS_ACCEPT`, then association establishment has completed. If the call is successful, but the `rtc_result` element is not `RTS_ACCEPT`, the the association attempt has been rejected; consult Table 4.1 to determine the reason for the reject. Otherwise, if the call fails then the association is not established and the `RtSAPabort` structure of the `RtSAPindication` discriminated union has been updated.

Note that the `rtc_data` element is allocated via *malloc*(3) and should be released using the `RTCFREE` macro when no longer referenced. The `RTCFREE` macro behaves as if it was defined as:

```
void    RTCFREE (rtc)
struct RtSAPconnect *rtc;
```

The macro frees only the data allocated by `RtBeginRequest`, and not the `RtSAPconnect` structure itself. Further, `RTCFREE` should be called only if the call to the `RtBeginRequest` routine returned `OK`.

A.2.3 Association Release

The `RtEndRequest` routine is used to request the release of an association, and corresponds to a X.410 CLOSE.REQUEST action. This action may be taken by only the initiator of an association, and, if the dialogue mode is two-way alternate, if the initiator has the turn as well.

```
int     RtEndRequest (sd, rti)
int     sd;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd`: the association-descriptor; and,

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

If the call to `RtEndRequest` is successful, then the association has been released. Otherwise if the call fails and the error is not fatal, then the association remains established and the `RtSAPabort` structure contained in the `RtSAPindication` parameter `rti` contains the reason for the failure.

Upon receiving a X.410 CLOSE.INDICATION event, the user is required to generate a X.410 CLOSE.RESPONSE action using the `RtEndResponse` routine.

```
int     RtEndResponse (sd, rti)
int     sd;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd`: the association-descriptor; and,

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

If the call to `RtEndResponse` is successful, then the association has been released.

A.2.4 An Example

Let's consider how one might construct a generic server that uses reliable transfer services to establish an association, but then uses remote operation services to communicate with its peer. This entity will use a synchronous interface.

There are two parts to the program: initialization and the request/reply loop. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure A.5, the initialization steps for the generic responder, including the outer *C* wrapper, is shown. First, the RtSAP state is re-captured by calling `RtBInit`. If this succeeds, then the association is authenticated and any command line arguments are parsed. Assuming that the responder is satisfied with the proposed association, it calls `RtBeginResponse` to accept the association. The `RoSetService` routine is called to set the underlying service to be used for remote operations. Finally, the main request/reply loop is realized. The responder calls `RoWaitRequest` to get the next event, and then calls `ros_indication` to decode that event.

Figure 3.2 on page 77 contains the definition of the `ros_indication` routine and associated routines. Since the reliable transfer service was used to establish the association, a different closing handler must be used. This is shown in Figure A.6.

```
#include <stdio.h>
#include <isode/rtsap.h>
#include <isode/isoservent.h>


main (argc, argv, envp)
int     argc;
char  **argv,
      **envp;
{
    int     result,
            sd;
    struct RoSAPindication rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;
    struct RtSAPstart   rtss;
    register struct RtSAPstart *rts = &rtss;
    struct RtSAPindication  rtis;
    register struct RtSAPindication *rti = &rtis;
    register struct RtSAPabort   *rta = &rti -> rti_abort;

    if (RtInit (argc, argv, rts, rti) == NOTOK)
        error ("initialization fails:  %s", RtErrString (rta -> rta_reason));

    sd = rts -> rts_sd;
/*  do authentication using rts -> rts_data here... */
    RTSFREE (rts);

/* read command line arguments here... */

/* could use RTS_BUSY, RTS_VALIDATE, or RTS_MODE instead and then exit */

    if (RtBeginResponse (sd, RTS_ACCEPT, NULLPE, rti) == NOTOK)
        error ("RT-BEGIN.RESPONSE: %s", RtErrString (rta -> rta_reason));

    if (RoSetService (sd, RoPService, roi) == NOTOK)
        error ("RoSetService:  %s", RoErrString (rop -> rop_reason));

    for (;;)
        switch (result = RoWaitRequest (sd, NOTOK, roi)) {
            case NOTOK:
            case OK:
            case DONE:
                ros_indication (sd, roi);
                break;

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

Figure A.5: Initializing the generic RTS responder

```
...

static int  ros_end (sd, roe)
int     sd;
struct RoSAPend *roe;
{
    struct RtSAPindication  rtis;
    register struct RtSAPindication *rti = &rtis;
    register struct RtSAPabort    *rta = &rti -> rti_abort;

    if (RtEndResponse (sd, rti) == NOTOK)
        error ("RT-END.RESPONSE: %s", RtErrString (rta -> rta_reason));

    exit (0);
}
```

Figure A.6: Finalizing the generic RTS responder

Bibliography

[CCI84a] Message Handling Systems: Presentation Transfer Syntax and Notation. International Telegraph and Telephone Consultative Committee, October 1984. Recommendation X.409.

[CCI84b] Message Handling Systems: Remote Operations and Reliable Transfer Server. International Telegraph and Telephone Consultative Committee, October 1984. Recommendation X.410.

[CCI88a] Reliable Transfer: Model and Service Definition. International Telegraph and Telephone Consultative Committee, March 1988. Recommendation X.218.

[CCI88b] Reliable Transfer: Protocol Specification. International Telegraph and Telephone Consultative Committee, March 1988. Recommendation X.228.

[CCI88c] Remote Operations: Model, Notation and Service Definition. International Telegraph and Telephone Consultative Committee, March 1988. Recommendation X.219.

[CCI88d] Remote Operations: Protocol Specification. International Telegraph and Telephone Consultative Committee, March 1988. Recommendation X.229.

[CCI88e] Specification of Abstract Syntax Notation One. International Telegraph and Telephone Consultative Committee, 1988. Recommendation X.208.

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

[ISO87a] 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.

[ISO87b] Information Processing — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). International Organization for Standardization and International Electrotechnical Committee, 1987. International Standard 8825.

[ISO88a] Information Processing Systems — Open Systems Interconnection — Service Definition for the Association Control Service Element. International Organization for Standardization and International Electrotechnical Committee, April 1988. Revised Final Text of Draft International Standard 8649.

[ISO88b] Information Processing Systems — Open Systems Interconnection – Protocol Specification for the Association Control Service Element. International Organization for Standardization and International Electrotechnical Committee, April 1988. Revised Final Text of Draft International Standard 8650.

[ISO88c] Information Processing Systems — Text Communication — Reliable Transfer Part 1: Model and Service Definition. International Organization for Standardization and International Electrotechnical Committee, March 1988. Working Document for International Standard 9066-1.

[ISO88d] Information Processing Systems — Text Communication — Reliable Transfer Part 2: Protocol Specification. International Organization for Standardization and International Electrotechnical Committee, March 1988. Working Document for International Standard 9066-2.

[ISO88e] 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.

[ISO88f] 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.

[Kil89] Stephen E. Kille. A string encoding of Presentation Address. Research Note RN/89/14, Department of Computer Science, University College London, February 1989.

[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