The ISO Development Environment: User's Manual

Volume 2: Underlying 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 #6.19) 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 #6.19) 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

Underlying Services

Chapter 2

Presentation Services

The *libpsap2*(3n) library implements the presentation service. The kernel subset of the ISO specification is currently implemented. That is, the library supports whatever session requirements the user wishes to employ, negotiates presentation contexts on connection establishments, and utilizes abstract transfer notations to transmit data structures in a machine-independent manner.

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 `psap2` 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 *libpsap2*(3n) library are integer-valued. They return the manifest constant `OK` on success, or `NOTOK` otherwise.

2.1 Warning

Please read the following important message.

NOTE:	Readers of this chapter should have an intimate understanding of the OSI presentation and session services. It is not the intent of this chapter to present a tutorial on these services, so novice users will suffer greatly if they choose to read further.

2.2 Addresses

Addresses at the presentation service access point are represented by the `PSAPaddr` structure.

```
struct PSAPaddr {
    struct SSAPaddr pa_addr;

#define PSSIZE  64
    int     pa_selectlen;
    char    pa_selector[PSSIZE];
};
#define NULLPA  ((struct PSAPaddr *) 0)
```

This structure contains two elements:

`pa_addr`: the session address, as described in Section 3.2 on page 42; and,

`pa_selector/pa_selectlen`: the presentation selector.

In Figure 2.1, an example of how one constructs the PSAP address for the LOOP provider on host `RemoteHost` is presented. The routine `is2paddr` takes a host and service, and then consults the *isoentities*(5) file described in Chapter 7 of *Volume One* to construct a presentation address.

```
struct PSAPaddr *is2paddr (host, service, is)
char   *host,
       *service;
struct isoservent *is;
```

```
#include <isode/psap2.h>
#include <isode/isoservent.h>

...

register struct PSAPaddr *pa;
register struct isoservent *is;

...

if ((is = getisoserventbyname ("loop", "psap")) == NULL)
    error ("psap/loop");

/* RemoteHost is the host we're interested in,
   e.g., "gremlin.nrtc.northrop.com" */

if ((pa = is2paddr (RemoteHost, NULLCP, is)) == NULLPA)
    error ("address translation failed");

...
```

Figure 2.1: Constructing the PSAP address for the LOOP provider

2.2.1 Calling Address

Certain users of the presentation service (e.g., the reliable transfer service) need to know the name of the local host when they initiate a connection. The routine `PLocalHostName` has been provided for this reason.

```
char   *PLocalHostName ()
```

2.3 Connection Establishment

Until the connection has been fully established, the implementation distinguishes between clients and servers, which are more properly referred to as *initiators* and *responders*, to use OSI terminology.

2.3.1 Connection Negotiation

From the user's perspective, there are several parameters which are negotiated by the presentation providers during connection establishment. Suggestions as to the values of some of these parameters are made by the user.

Session Parameters

Consult Section 3.3.1 for a list of session parameters which are negotiated during connection establishment.

Presentation Contexts

A *presentation context* is a binding between an abstract syntax notation and an abstract transfer notation on a presentation connection, and is denoted by an integer value termed the context identifier. The abstract syntax notation describes, to the users of the presentation service, the data structures being exchanged; the abstract transfer notation describes, to the providers of the presentation services, the method for encoding those data structures in a machine-independent fashion. Hence the abstract syntax notation is negotiated by the users of the presentation service, while the abstract transfer notation is negotiated by the providers of that service.

When a connection is established, the initiator suggests zero or more presentation contexts, specifying a context identifier (an odd-valued integer), and the abstract syntax (a pointer to an object identifier, see Section 5.4.6 of *Volume One*). The provider selects the abstract transfer notation (in the current implementation, this is always ASN.1[ISO87a]). When the a P-CONNECT.INDICATION event is given to the responder, in addition indicating the context identifier and abstract syntax information, the provider also indicates if it is willing to support this presentation context. If so, the responder decides if it will accept or reject the context. This information is then propagated back to the initiator with the P-CONNECT.CONFIRMATION indication.

2.3.2 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 presentation service, the *tsapd* program contains the bootstrap for the presentation provider. The daemon will again consult the ISO services database to determine which program on the system implements the desired PSAP 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 PSAP state contained in the magic arguments. This is done by calling the routine `PInit`, which on a successful return, is equivalent to a P-CONNECT.INDICATION from the presentation service provider.

```
int     PInit (vecp, vec, ps, pi)
int     vecp;
char  **vec;
struct PSAPstart *ps;
struct PSAPindication *pi;
```

The parameters to this procedure are:

`vecp:` the length of the argument vector;

`vec:` the argument vector;

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

pi: a pointer to a PSAPindication structure, which is updated only if the call fails.

If PInit is successful, it returns information in the ps parameter, which is a pointer to a PSAPstart structure.

```
struct PSAPstart {
    int     ps_sd;

    struct PSAPaddr ps_calling;
    struct PSAPaddr ps_called;

    struct PSAPctxlist ps_ctxlist;

    OID     ps_defctx;
    int     ps_defctxresult;

    int     ps_prequirements;

    int     ps_srequirements;
    int     ps_settings;
    long    ps_isn;

    struct SSAPref  ps_connect;

    int     ps_ssdusize;

    struct QOStype ps_qos;

#define NPDATA        10
    int     ps_ninfo;
    PE      ps_info[NPDATA];
};
```

The elements of this structure are:

ps_sd: the presentation-descriptor to be used to reference this connection;

ps_calling: the address of the peer initiating the connection;

ps_called: the address of the peer being asked to respond;

ps_ctxlist: the proposed list of presentation contexts;

ps_defctx/ps_defctxresult: the default context for the presentation connection (and the presentation provider's response);

ps_prequirements: the proposed presentation requirements;

ps_srequirements: the proposed session requirements;

ps_settings: the initial token settings;

ps_isn: the initial serial-number;

ps_connect: the connection identifier (a.k.a. SSAP reference) used by the initiator;

ps_ssdusize: the largest atomic SSDU size that can be used on the connection (see the note on page 45);

ps_qos: the quality of service on the connection (see Section 4.6.2); and,

ps_info/ps_ninfo: an array of initial data (and the number of elements in that array).

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

```
void    PSFREE (ps)
struct PSAPstart *ps;
```

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

The `ps_connect` element is a `SSAPref` structure, which is passed transparently by the presentation service. Consult the description on page 52. There are two routines of interest in the *libpsap2*(3n) that deal with these structures: The `addr2ref` routine takes a string (presumably a hostname), determines the current UT time, and returns a pointer to a `SSAPref` structure appropriately initialized to denote this information.

```
struct SSAPref *addr2ref (addr)
char   *addr;
```

This routine might fail if it is unable to allocate a small amount of memory. In this event, it returns the manifest constant `NULL`. The routine `sprintref` can be used to return a null-terminated string describing the SSAP reference.

```
char   *sprintref (sr)
struct SSAPref *sr;
```

The `ps_ctxlist` element is a `PSAPctxlist` structure, which describes the presentation contexts the initiator wishes to use.

```
struct PSAPctxlist {
    int     pc_nctx;

#define NPCTX   5

    struct PSAPcontext {
        int     pc_id;
        OID     pc_asn;
        OID     pc_atn;
        int     pc_result;
    }       pc_ctx[NPCTX];
};
#define NULLPC  ((struct PSAPctxlist *) 0)
```

The elements of this structure are:

`pc_ctx/pc_nctx`: the presentation contexts described (and the number of contexts which may not exceed `NPCTX` elements). For each presentation context:

- `pc_id`: the identifier (or handle) for the context;
- `pc_asn`: the abstract syntax notation to be used on the context;
- `pc_atn`: the transfer syntax notation to be used on the context (this field is usually, `NULLOID`, only the initiator, when it wishes to inform the presentation service of the transfer syntax to use, is permitted to specify this); and,

`pc_result`: the presentation provider's response (codes are listed in Table 2.1).

If the call to the `PInit` routine is not successful, then a P-P-ABORT.INDICATION event is simulated, and the relevant information is returned in a `PSAPindication` structure.

```
struct PSAPindication {
    int     pi_type;
#define PI_DATA         0x00
#define PI_TOKEN        0x01
#define PI_SYNC         0x02
#define PI_ACTIVITY     0x03
#define PI_REPORT       0x04
#define PI_FINISH       0x05
#define PI_ABORT        0x06

    union {
        struct PSAPdata pi_un_data;
        struct PSAPtoken pi_un_token;
        struct PSAPsync pi_un_sync;
        struct PSAPactivity pi_un_activity;
        struct PSAPreport pi_un_report;
        struct PSAPfinish pi_un_finish;
        struct PSAPabort pi_un_abort;
    }   pi_un;
#define pi_data         pi_un.pi_un_data
#define pi_token        pi_un.pi_un_token
#define pi_sync         pi_un.pi_un_sync
#define pi_activity     pi_un.pi_un_activity
#define pi_report       pi_un.pi_un_report
#define pi_finish       pi_un.pi_un_finish
#define pi_abort        pi_un.pi_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 `PSAPabort` structure contained therein, and then consults the elements of that structure.

```
struct PSAPabort {
    int     pa_peer;

    int     pa_reason;
    int     pa_ppdu;


    int     pa_ninfo;
    char    pa_info;

#define PA_SIZE        512
    int     pa_cc;
    char    pa_data[PA_SIZE];
};
```

The elements of a `PSAPabort` structure are:

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

`pa_reason`: the reason for the provider-initiated abort (codes are listed in Table 2.1);

`pa_ppdu`: the type of presentation protocol data unit which triggered the provider-initiated abort (codes are listed in Table 2.2);

`pa_data/pa_cc`: a provider-generated diagnostic string (and the length of that string); and,

`pa_info/pa_ninfo`: an array of user data, and the number of elements in that array (if `pa_peer` is set).

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

```
void    PAFREE (pa)
struct PSAPabort *pa;
```

Provider-initiated Abort (fatal)	
`PC_ADDRESS`	Called presentation address unknown
`PC_AVAILABLE`	No PSAP available from those identified presentation address
`PC_CONGEST`	Local limit exceeded
`PC_VERSION`	Protocol version not supported
`PC_UNSPECIFIED`	Unspecified
`PC_UNRECOGNIZED`	Unrecognized PPDU
`PC_UNEXPECTED`	Unexpected PPDU
`PC_SSPARAM`	Unexpected session service parameter
`PC_PPPARAM`	Unrecognized PPDU parameter
`PC_INVALID`	Invalid PPDU parameter
`PC_ABSTRACT`	Abstract syntax not supported
`PC_TRANSFER`	Proposed transfer syntaxes not supported
`PC_REFUSED`	Connect request refused on this network connection
`PC_SESSION`	Session disconnect
`PC_PROTOCOL`	Protocol error
`PC_ABORTED`	Peer aborted connection
User-initiated Abort (fatal)	
`PC_REJECTED`	Rejected by peer
Interface Errors (non-fatal)	
`PC_PARAMETER`	Invalid parameter
`PC_OPERATION`	Invalid operation
`PC_TIMER`	Timer expired
`PC_WAITING`	Indications waiting

Table 2.1: PSAP Failure Codes

PPDU	**Associated Operation**
PPDU_NONE	none
PPDU_CP	connection
PPDU_CPA	connection accept
PPDU_CPR	connection reject
PPDU_ARU	user abort
PPDU_ARP	provider abort
PPDU_AC	alter context
PPDU_ACA	alter context ack
PPDU_TD	data
PPDU_TTD	typed-data
PPDU_TE	expedited data
PPDU_TC	capability data
PPDU_TCC	capability data ack
PPDU_RS	resynchronize
PPDU_RSA	resynchronize ack

Table 2.2: PSAP PPDU Codes

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

After examining the information returned by PInit on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the connection. For either response, the responder should use the PConnResponse routine (which corresponds to the P-CONNECT.RESPONSE action).

```
int     PConnResponse (sd, status, responding, ctxlist,
                defctxresult, prequirements, srequirements,
                isn, settings, ref, data, ndata, pi)
int     sd;
struct PSAPaddr *responding;
int     status,
        prequirements,
        srequirements,
        settings,
        ndata;
long isn;
struct PSAPctxlist *ctxlist;
int     defctxresult;
struct SSAPref *ref;
PE      *data;
struct PSAPindication *pi;
```

The parameters to this procedure are:

sd: the presentation-descriptor;

result: the acceptance indicator (either PC_ACCEPT if the connection is to be accepted, or PC_REJECTED otherwise);

responding: the PSAP address of the responder (defaulting to the called address, if not present);

ctxlist: the responder's decision as to each of the presentation contexts suggested (for each proposed context, if the pc_result element supplied by the provider is PC_ACCEPT, which indicates that the provider is willing to support it, the user may supply either PC_ACCEPT or the value PC_REJECTED);

`defctxresult`: the response to the default context (if the presentation provider responded with `PC_ACCEPT`, the user may supply either `PC_ACCEPT` or `PC_REJECTED`);

`prequirements`: the responder's presentation requirements;

`srequirements`: the responder's session requirements;

`isn`: the initial serial-number;

`settings`: the initial token settings;

`ref`: the connection identifier used by the responder (consult page 52 for a description of the `SSAPref` structure);

`data/ndata`: an array of initial data (and the number of elements in that array, consult the warning on page 35); and,

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

If the call to `PConnResponse` is successful, and if the `result` parameter is set to `PC_ACCEPT`, then connection establishment has completed and the users of the presentation service now operate as symmetric peers. If the call is successful, but the `result` parameter is `PC_REJECTED` instead, then the connection has been rejected and the responder may exit. Otherwise, if the call fails and the reason is not an interface error (see Table 2.1 on page 25), then the connection is closed.

Note that when the responder rejects the connection, it need only supply meaningful values for the `sd`, `status`, `defctxresult`, and `pi` parameters.

2.3.3 Client Initialization

A program wishing to connect to another user of presentation services calls the `PConnRequest` routine, which corresponds to the P-CONNECT.REQUEST action.

```
int     PConnRequest (calling, called, ctxlist, defctxname,
                prequirements, srequirements, isn, settings,
                ref, data, ndata, qos, pc, pi)
```

```
struct PSAPaddr *calling,
                *called;
int     prequirements,
        srequirements,
        settings,
        ndata;
long isn;
struct PSAPctxlist *ctxlist;
OID     defctxname;
struct SSAPref *ref;
PE    *data;
struct QOStype *qos;
struct PSAPconnect *pc;
struct PSAPindication *pi;
```

The parameters to this procedure are:

`calling`: the PSAP address of the initiator (use the manifest constant verb"NULLPA" if this is not unimportant);

`called`: the PSAP address of the responder;

`ctxlist`: the list of proposed presentation contexts (only the `pc_id`, `pc_asn`, and optionally the `pc_atn` elements should be filled in);

`defctxname`: the proposed default contexts;

`prequirements`: the presentation requirements;

`srequirements`: the session requirements;

`isn`: the initial serial-number;

`settings`: the initial token settings;

`ref`: the connection identifier used by the initiator (consult page 52 for a description of the `SSAPref` structure);

`data/ndata`: an array of initial data (and the number of elements in that array, consult the warning on page 35);

`qos`: the quality of service on the connection (see Section 4.6.2);

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

pi: a pointer to a PSAPindication structure, which is updated only if the call fails.

If the call to PConnRequest is successful (a successful return corresponds to a P-CONNECT.CONFIRMATION event), then information is returned in the pc parameter, which is a pointer to a PSAPconnect structure.

```
struct PSAPconnect {
    int     pc_sd;

    struct PSAPaddr pc_responding;

    struct PSAPctxlsit pc_ctxlist;

    int     pc_defctxresult;

    int     pc_prequirements;

    int     pc_srequirements;
    int     pc_settings;
    int     pc_please;
    long    pc_isn;

    struct SSAPref pc_connect;

    int     pc_ssdusize;

    struct QOStype pc_qos;
    int     pc_result;

    struct SSAPref  pc_connect;

#define PC_SIZE         512
    int     pc_cc;
    char    pc_data[PC_SIZE];
};
```

The elements of this structure are:

`pc_sd`: the presentation-descriptor to be used to reference this connection;

`pc_responding`: the responding peer's address (which is the same as the `called` parameter given to `SConnRequest`);

`pc_ctxlist`: the (negotiated) list of presentation contexts;

`pc_defctxresult`: the response to the proposed default context;

`pc_prequirements`: the (negotiated) presentation requirements;

`pc_srequirements`: the (negotiated) session requirements;

`pc_settings`: the (negotiated) initial token settings;

`pc_please`: the tokens which the responder wants to own (if any);

`pc_isn`: the (negotiated) initial serial-number;

`pc_connect`: the connection identifier used by the responder (consult page 52 for a description of the `SSAPref` structure);

`pc_ssdusize`: the largest atomic SSDU size that can be used on the connection (see the note on page 45);

`pc_qos`: the quality of service on the connection (see Section 4.6.2); and,

`pc_result`: the connection response;

`pc_info/pc_ninfo`: an array of initial data, and the number of elements in that array.

If the call to `PConnRequest` is successful, and the `pc_result` element is set to `PC_ACCEPT`, then connection establishment has completed and the users of the presentation service now operate as symmetric peers. If the call is successful, but the `pc_result` element is not `PC_ACCEPT`, then the connection has been rejected; consult Table 2.1 to determine the reason (further information can be found in the `pi` parameter). Otherwise, if the call fails then the connection is not established and the `PSAPabort` structure of the `PSAPindication` discriminated union has been updated.

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

```
void    PCFREE (pc)
struct PSAPconnect *pc;
```

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

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

```
int     PAsynConnRequest (calling, called, ctxlist,
                defctxname, prequirements, srequirements,
                isn, settings, ref, data, ndata, qos, pc,
                pi, async)
struct PSAPaddr *calling,
                *called;
int     prequirements,
        srequirements,
        settings,
        ndata,
        async;
long isn;
struct PSAPctxlist *ctxlist;
OID     defctxname;
struct SSAPref *ref;
PE    *data;
struct QOStype *qos;
struct PSAPconnect *pc;
struct PSAPindication *pi;
```

The additional parameter to this procedure is:

async: whether the connection should be initiated asynchronously.

If the async parameter is non-zero, then PAsynConnRequest returns one of four values: NOTOK, which indicates that the connection request failed; DONE, which indicates that the connection 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 PConnRequest are employed (i.e., a NOTOK return from PAsynConnRequest is equivalent to a NOTOK return from PConnRequest, and, a DONE return from PAsynConnRequest is equivalent to a OK return from PConnRequest).

In the final case, when either CONNECTING_1 or CONNECTING_2 is returned, only the pc_sd element of the pc parameter has been updated; it reflects the presentation-descriptor to be used to reference this connection.

To determine when the connection attempt has been completed, the routine xselect (consult Section 2.4 of *Volume One*) should be used after calling PSelectMask. In order to determine if the connection attempt was successful, the routine PAsynRetryRequest is called:

```
int     PAsynRetryRequest (sd, pc, pi)
int     sd;
struct PSAPconnect *pc;
struct PSAPindication  *pi;
```

The parameters to this procedure are:

sd: the presentation-descriptor;

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

pi: a pointer to a PSAPindication structure, which is updated only if the call fails.

Again, one of three values are returned: NOTOK, which indicates that the connection request failed; DONE, which indicates that the connection 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 for information on how to make efficient use of the asynchronous connection facility.

2.4 Data Transfer

Once the connection has been established, a presentation-descriptor is used to reference the connection. This is usually the first parameter given to any of the remaining routines in the *libpsap2*(3n) library. Further, the last parameter is usually a pointer to a `PSAPindication` structure (as described on page 23). If a call to one of these routines fails, then the structure is updated. Consult the `PSAPabort` element of the `PSAPindication` structure. If the `pa_reason` element of the `PSAPabort` structure is associated with a fatal error, then the connection is closed. That is, a P-P-ABORT.INDICATION event has occurred. The `PC_FATAL` macro can be used to determine this.

```
int     PC_FATAL (r)
int     r;
```

The most common interface error to occur is `PC_OPERATION` which usually indicates that either the user is lacking ownership of a session token to perform an operation, or that a session requirement required by the operation was not negotiated during connection establishment. For protocol purists, the `PC_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     PC_OFFICIAL (r)
int     r;
```

All of the remaining routines in the *libpsap2*(3n) library are identical to their counterparts in the *libssap*(3n) library, with these exceptions:

- The final parameter to each routine is a pointer to a `PSAPindication` structure, rather than a `SSAPindication` structure.

- Any user data components are specified as an array of presentation elements (and the number of elements in that array), rather than the base of a character array (and the number of octets to be sent). Note that any data to be sent should have the `pe_context` element set to the desired presentation context. By default, presentation elements are initialized with the default context (as represented by the manifest constant `PE_DFLT_CTX`).

- Asynchronous event handlers are called with pointers to `PSAP` structures, rather tha `SSAP` structures.
- With any indication which occurs, it is important to free any data which might have been allocated. The structures and corresponding macros are:

Macro	Structure Pointer
PXFREE	struct PSAPdata *
PTFREE	struct PSAPtoken *
PNFREE	struct PSAPsync *
PVFREE	struct PSAPactivity *
PPFREE	struct PSAPreport *
PFFREE	struct PSAPfinish *
PRFREE	struct PSAPrelease *
PAFREE	struct PSAPabort *

  Note that these free the user data referenced by the indication structures, and not the structures themselves.

2.4.1 Restrictions on User Data

To quote the [ISO88a] specification:

> **NOTE:** For all services which carry user data, excluding P-DATA and P-TYPED-DATA, it may not be possible to exchange user data, dependent on the user data length limitation supported by the underlying session services.

2.5 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `pi` parameter given to the routine. The `pi_abort` element of the `PSAPindication` structure contains a `PSAPabort` structure detailing the reason for the failure. The `pa_reason` element of this latter structure can be given as a parameter to the routine `PErrString` which returns a null-terminated diagnostic string.

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

2.6 Compiling and Loading

Programs using the *libpsap2*(3n) library should include `<isode/psap2.h>`. The programs should also be loaded with `-lpsap2`.

2.7 An Example

Let's consider how one might construct a source entity that resides on the PSAP. This entity will use a synchronous interface.

There are two parts to the program: initialization and data transfer; release will occur when the standard input has been exhausted. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure 2.2, the initialization steps for the source entity, including the outer *C* wrapper, is shown. First, a lookup is done in the ISO services database, and the `PSAPaddr` is initialized. The `SSAPref` is initialized using the routine `addr2ref`. This routine takes a string and returns a pointer to a `SSAPref` structure which has been initialized to contain the string and the current UTC time. Next, for each token associated with the session requirements, initial ownership of that token is claimed. Finally, the call to `PConnRequest` is made. If the call is successful, a check is made to see if the remote user accepted the connection. If so, the presentation-descriptor is captured, along with the negotiated requirements and initial token settings.

In Figure 2.3 on page 39, the data transfer loop is realized. The source entity reads a line from the standard input, and then queues that line for sending to the remote side. When an end-of-file occurs on the standard input, the source entity requests release and then gracefully terminates. Although no checking is done in this example, for the calls to `PDataRequest` and `PRelRequest`, on failure a check for the operational error `PC_OPERATION` should be made. For `PDataRequest`, this would occur when the data token was not owned by the user; for `PRelRequest`, this would occur when the release token was not owned by the user.

```
#include <stdio.h>
#include <isode/psap2.h>
#include <isode/isoservent.h>


static int requirements = SR_HALFDUPLEX | SR_NEGOTIATED;

static int owned = 0;


/* ARGSUSED */

main (argc, argv, envp)
int     argc;
char  **argv,
      **envp;
{
    int     sd,
            settings;
    char    buffer[BUFSIZ];
    register struct PSAPaddr   *pz;
    register struct SSAPref *sf;
    struct PSAPconnect  pcs;
    register struct PSAPconnect *pc = &pcs;
    struct PSAPrelease  prs;
    register struct PSAPrelease *pr = &prs;
    struct PSAPindication   pis;
    register struct PSAPindication *pi = &pis;
    register struct PSAPabort *pa = &pi -> pi_abort;
    register struct isoservent *is;

    if ((is = getisoserventbyname ("sink", "psap")) == NULL)
        error ("psap/sink:  unknown provider/entity pair");
    if (argc != 2)
        error ("usage:  source \"host\"");

...
```

Figure 2.2: Initializing the PSAP source entity

```
...

    if ((pz = is2paddr (argv[1], NULLCP, is)) == NULLPA)
            error ("address translation failed");
    sf = addr2ref (PLocalHostName ());

    settings = 0;
#define dotoken(requires,shift,bit,type) \
{ \
    if (requirements & requires) \
            settings |= ST_INIT_VALUE << shift; \
}
    dotokens ();
#undef dotoken

    if (PConnRequest (NULLSA, pz, NULLPC, NULLOID, 0, requirements,
                SERIAL_NONE, settings, sf, NULLPEP, 0, NULLQOS, pc, pi) == NOTOK)
            error ("P-CONNECT.REQUEST: %s", PErrString (pa -> pa_reason));
    if (pc -> pc_result != PC_ACCEPT)
            error ("connection rejected by sink: %s",
                        PErrString (pc -> pc_result));

    sd = pc -> pc_sd;
    requirements = pc -> pc_requirements;

#define dotoken(requires,shift,bit,type) \
{ \
    if (requirements & requires) \
            if ((pc -> pc_settings & (ST_MASK << shift)) \
                        == ST_INIT_VALUE) \
                owned |= bit; \
}
            dotokens ();
#undef dotoken

    PCFREE (pc);

...
```

Figure 2.2: Initializing the PSAP source entity (continued)

```
...

    while (fgets (buffer, sizeof buffer, stdin)) {
            register PE = oct2prim (buffer, strlen (buffer) + 1);

            if (PDataRequest (sd, &pe, 1, pi) == NOTOK)
                error ("P-DATA.REQUEST: %s", PErrString (pa -> pa_reason));

            pe_free (pe);
    }

    if (PRelRequest (sd, NULLPEP, 0, NOTOK, pr, pi) == NOTOK)
            error ("P-RELEASE.REQUEST: %s", PErrString (pa -> pa_reason));

    if (!pr -> pr_affirmative) {
            (void) PUAbortRequest (sd, NULLPEP, 0, pi);
            error ("release rejected by sink");
    }
    PRFREE (pr);

    exit (0);
}
```

Figure 2.3: Data Transfer for the PSAP source entity

2.8 Lightweight Presentation Protocol

To run OSI applications using the lighweight presentation protocol defined in RFC1085, load the program with `-lpsap2-lpp` or `-isode-lpp`.

This is a complete implementation of RFC1085. The following functions are available:

```
PInit
PConnResponse
PAsynConnRequest
PAsynRetryRequest
PDataRequest
PReadRequest
```

```
PUAbortRequest
PRelRequest
PRelResponse
PSetIndications
PSelectMask
PErrSTring
```

Note that when RFC1085 is used as a presentation backing, only a subset of the presentation services are available:

P-CONNECT P-DATA P-U-ABORT P-P-ABORT

The *lppd*(8c) daemon is used to listen for incoming connections and dispatch the appropriate daemon. This daemon will listen only for connections using the TCP backing. For connections using the UDP backing, the responder program must listen itself. This is trivally accomplished using the `isodeserver` routine described in Section 2.5 on page 44.

2.9 For Further Reading

The ISO specification for session services is defined in [ISO88a], while the corresponding protocol definition is [ISO88b].

Chapter 3

Session Services

The *libssap*(3n) library implements the session service.

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 `ssap` 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 *libssap*(3n) library are integer-valued. They return the manifest constant `OK` on success, or `NOTOK` otherwise.

3.1 Warning

Please read the following important message.

NOTE:	Readers of this chapter should have an intimate understanding of the OSI session service. It is not the intent of this chapter to present a tutorial on these services, so novice users will suffer greatly if they choose to read further.

3.2 Addresses

Addresses at the session service access point are represented by the `SSAPaddr` structure.

```
struct SSAPaddr {
    struct TSAPaddr sa_addr;

#define SSSIZE  64
    int     sa_selectlen;
    char    sa_selector[SSSIZE];
};
#define NULLSA  ((struct SSAPaddr *) 0)
```

This structure contains two elements:

`sa_addr`: the transport address, as described in Section 4.1 on page 101; and,

`sa_selector/sa_selectlen`: the session selector.

In Figure 3.1, an example of how one constructs the SSAP address for the Presentation provider on host `RemoteHost` is presented. The routine `is2saddr` takes a host and service, and then consults the *isoentities*(5) file described in Chapter 7 of *Volume One* to construct a session address.

```
struct SSAPaddr *is2saddr (host, service, is)
char   *host,
       *service;
struct isoservent *is;
```

```
#include <isode/ssap.h>
#include <isode/isoservent.h>

...

register struct SSAPaddr *sa;
register struct isoservent *is;

...

if ((is = getisoserventbyname ("presentation", "ssap")) == NULL)
    error ("ssap/presentation");

/* RemoteHost is the host we're interested in,
   e.g., "gremlin.nrtc.northrop.com" */

if ((sa = is2saddr (RemoteHost, NULLCP, is)) == NULLSA)
    error ("address translation failed");

...
```

Figure 3.1: Constructing the SSAP address for the Presentation provider

3.2.1 Calling Address

Certain users of the session service (e.g., the reliable transfer service) need to know the name of the local host when they initiate a connection. The routine `SLocalHostName` has been provided for this reason.

```
char   *SLocalHostName ()
```

3.2.2 Address Encodings

It may be useful to encode a session address for viewing. Although a consensus for a standard way of doing this has not yet been reached, the routines `saddr2str` and `str2saddr` may be used in the interim.

```
char   *saddr2str (sa)
struct SSAPaddr *sa;
```

The parameter to this procedure is:

`sa`: the session address.

If `saddr2str` fails, it returns the manifest constant `NULLCP`.

The routine `str2saddr` takes an ascii string encoding and returns a session address.

```
struct SSAPaddr *str2saddr (str)
char   *str;
```

The parameter to this procedure is:

`str`: the ascii string.

If `str2saddr` fails, it returns the manifest constant `NULLSA`.

3.3 Connection Establishment

Until the connection has been fully established, the implementation distinguishes between clients and servers, which are more properly referred to as *initiators* and *responders*, to use OSI terminology.

3.3.1 Connection Negotiation

From the user's perspective, there are several parameters which are negotiated by the session providers during connection establishment. Suggestions as to the values of some of these parameters are made by the user.

Maximum SSDU Size

The session provider will accept arbitrarily large session service data units (SSDUs) and transparently fragment and re-assemble them during transit. Hence, the actual SSDU is of unlimited size. However, for efficiency reasons, it may be desirable for the user to send SSDUs which are no larger than a certain threshold. When a connection has been established, the service providers inform the initiator and responder as to what this threshold is.

NOTE:	In the current implementation, SSDUs which are no larger than the maximum atomic SSDU size are handled very efficiently. For optimal performance, users of the session service should strive to avoid sending SSDUs which are larger than this threshold.

Session Requirements

Users may specify the particular services that they will require from the session provider. The particular requirements supported in the current implementation are listed in Table 3.1. These requirements are always negotiated downward. That is, the initiator of the connection (i.e., the "client") indicates the desired session requirements. These are then given to the responder to the connection request (i.e., the "server") who may select any (or all) of the indicated session requirements.[1] This selection is then indicated to the initiator.

[1]There is one exception, the responder may not select both the half- and full-duplex requirements. It must choose one. If the initiator selects both initially, it is indicating that the choice is made at the responder's discretion.

Requirements	
SR_HALFDUPLEX	Half-duplex
SR_DUPLEX	Full-duplex
SR_EXPEDITED	Expedited Data Transfer
SR_MINORSYNC	Minor Synchronize
SR_MAJORSYNC	Major Synchronize
SR_RESYNC	Resynchronize
SR_ACTIVITY	Activity Management
SR_NEGOTIATED	Negotiated Release
SR_CAPABILITY	Capability Data Transfer
SR_EXCEPTIONS	Exception Reporting
SR_TYPEDATA	Typed Data Transfer
Subsets (combinations of the above)	
SR_BASUBSET	Basic Activity Subset
SR_BCSUBSET	Basic Combined Subset
SR_BSSUBSET	Basic Synchronized Subset

Table 3.1: Session Requirements

Token	Name	Availability
ST_RLS_TOKEN	release token	SR_RLS_EXISTS
ST_MAJ_TOKEN	majorsync token	SR_MAJ_EXISTS
ST_ACT_TOKEN	activity token (really majorsync token)	SR_ACT_EXISTS
ST_MIN_TOKEN	minorsync token	SR_MIN_EXISTS
ST_DAT_TOKEN	data token	SR_DAT_EXISTS

Table 3.2: Session Tokens

Session Tokens

Depending on the session requirements selected, several session tokens may be available in order to coordinate the use of session services.

There are two terms commonly used when referring to a session token:

- Availability
  If a token is available, then it exists for use during the session connection. The availability of a token depends on the session requirements selected for the connection.
- Ownership
  Certain session services are may not be requested by a user unless that user owns the token associated with those services.

The particular tokens supported in the current implementation, along with their associated availability information are listed in Table 3.2.

The session requirements are encoded as a single integer (actually, only the low-order 2 octets of an integer). To determine if a token is available, one can use a simple test involving the session requirements:

```
if (requirements & SR_xxx_EXISTS) {
    ...
}
```

For example, to determine if the negotiated release token is available:

```
if (requirements & SR_RLS_EXISTS) {
    ...
}
```

ST_INIT_VALUE	initiator's side
ST_RESP_VALUE	responder's side
ST_CALL_VALUE	responder's choice

Table 3.3: Initial Token Settings

Finally, the macro `dotokens` may be used to execute *C* code for each session token (regardless of availability). This macro acts as if it executes:

```
dotoken (SR_xxx_EXISTS, ST_xxx_SHIFT, ST_xxx_TOKEN, "xxx");
```

for each token. Usually, `dotoken` is a macro which executes some code for each token. An example is provided momentarily.

Initial Token Settings

For each token which is made available during connection negotiation, the choice as to which user initially has the token is left to the discretion of the initiator. The three choices for the initial settings of a token are listed in Table 3.3.

The initial settings for all available tokens are encoded in a single integer (actually, only the low-order 2 octets of an integer). To encode a value:

```
settings &= ~(ST_MASK << ST_yyy_SHIFT);
settings |= ST_xxx_VALUE << ST_yyy_SHIFT;
```

For example, to indicate that the responder of the connection is to initially have the data token:

```
settings &= ~(ST_MASK << ST_DAT_SHIFT);
settings |= ST_RESP_VALUE << ST_DAT_SHIFT;
```

The first statement, which clears the field in `settings` by using `ST_MASK`, is not required if `settings` is initially `0`.

If the initiator indicates that the initial setting of a token is left to the responder's choice, then the responder must decide. In Figure 3.2, an example of the `dotokens` macro is presented. In this example, the responder examines the initial setting for each available token, and:

- Notes if the responder initially owns the token (the `ST_RESP_VALUE` case); or,

- Gives ownership of the token to the initiator if the choice is at the responder's discretion (the `ST_CALL_VALUE` case).

3.3.2 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 TSAP entity. In the case of the session service, the *tsapd* program contains the bootstrap for the session provider. The daemon will again consult the ISO services database to determine which program on the system implements the desired SSAP 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 SSAP state contained in the magic arguments. This is done by calling the routine `SInit`, which on a successful return, is equivalent to a S-CONNECT.INDICATION from the session service provider.

```
int     SInit (vecp, vec, ss, si)
int     vecp;
char  **vec;
struct SSAPstart *ss;
struct SSAPindication *si;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

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

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

```
#include <ssap.h>

...

int     owned = 0;
int     required;               /* initialized from connection negotation */
int     settings;               /*    .. */

...

#define dotoken(requires,shift,bit,type) \
{ \
    if (requirements & requires) \
        switch (settings & (ST_MASK << shift)) { \
            case ST_CALL_VALUE << shift: \
                settings &= ~(ST_MASK << shift); \
                settings |= ST_INIT_VALUE << shift; \
                break; \
 \
            case ST_INIT_VALUE: \
                break; \
 \
            case ST_RESP_VALUE: \
                owned |= bit; \
                break; \
 \
            default: \
                error ("initial %s token setting", type); \
        } \
}

    dotokens ();

#undef dotoken
```

Figure 3.2: Determining the Initial Token Settings

If `SInit` is successful, it returns information in the `ss` parameter, which is a pointer to a `SSAPstart` structure.

```
struct SSAPstart {
    int     ss_sd;

    struct SSAPref  ss_connect;

    struct SSAPaddr ss_calling;
    struct SSAPaddr ss_called;

    int     ss_requirements;
    int     ss_settings;
    long    ss_isn;

    int     ss_ssdusize;

    struct QOStype ss_qos;

#define SS_SIZE         512
    int     ss_cc;
    char    ss_data[SS_SIZE];
};
```

The elements of this structure are:

`ss_sd`: the session-descriptor to be used to reference this connection;

`ss_connect`: the connection identifier (a.k.a. SSAP reference) used by the initiator;

`ss_calling`: the address of the peer initiating the connection;

`ss_called`: the address of the peer being asked to respond;

`ss_requirements`: the proposed session requirements;

`ss_settings`: the initial token settings;

`ss_isn`: the initial serial-number;

`ss_ssdusize`: the largest atomic SSDU size that can be used on the connection (see the note on page 45);

`ss_qos`: the quality of service on the connection (see Section 4.6.2); and,

`ss_data/ss_cc`: any initial data (and the length of that data).

The `ss_connect` element is a `SSAPref` structure, which is passed transparently by the session service.

```
struct SSAPref {
#define SREF_USER_SIZE 64
    u_char sr_ulen;
    char    sr_udata[SREF_USER_SIZE];

#define SREF_COMM_SIZE 64
    u_char  sr_clen;
    char    sr_cdata[SREF_COMM_SIZE];

#define SREF_ADDT_SIZE 4
    u_char  sr_alen;
    char    sr_adata[SREF_ADDT_SIZE];

    u_char  sr_vlen;
    char    sr_vdata[SREF_USER_SIZE];
};
```

The elements of this structure are:

`sr_udata/sr_ulen`: the user reference (and length of that reference, which may not exceed `SREF_USER_SIZE` octets);

`sr_cdata/sr_clen`: the common reference (and length of that reference, which may not exceed `SREF_COMM_SIZE` octets);

`sr_adata/sr_adata`: the additional reference (and length of that reference, which may not exceed `SREF_ADDT_SIZE` octets); and,

`sr_vdata/sr_vlen`: a second user reference (and length of that reference, which may not exceed `SREF_USER_SIZE` octets), which is used only when starting or resuming an activity.

If the call to the `SInit` routine is not successful, then a S-P-ABORT.INDICATION event is simulated, and the relevant information is returned in a `SSAPindication` structure.

```
struct SSAPindication {
    int     si_type;
#define SI_DATA         0x00
#define SI_TOKEN        0x01
#define SI_SYNC         0x02
#define SI_ACTIVITY     0x03
#define SI_REPORT       0x04
#define SI_FINISH       0x05
#define SI_ABORT        0x06

    union {
        struct SSAPdata si_un_data;
        struct SSAPtoken si_un_token;
        struct SSAPsync si_un_sync;
        struct SSAPactivity si_un_activity;
        struct SSAPreport si_un_report;
        struct SSAPfinish si_un_finish;
        struct SSAPabort si_un_abort;
    }   si_un;
#define si_data         si_un.si_un_data
#define si_token        si_un.si_un_token
#define si_sync         si_un.si_un_sync
#define si_activity     si_un.si_un_activity
#define si_report       si_un.si_un_report
#define si_finish       si_un.si_un_finish
#define si_abort        si_un.si_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 `SSAPabort` structure contained therein, and then consults the elements of that structure.

```
struct SSAPabort {
    int     sa_peer;
```

```
        int     sa_reason;

    #define SA_SIZE         512
        int     sa_cc;
        char    sa_data[SA_SIZE];
    };
```

The elements of a SSAPabort structure are:

sa_peer: if set, indicates that a user-initiated abort occurred (a S-U-ABORT.INDICATION event); if not set, indicates that a provider-initiated abort occurred (a S-P-ABORT.INDICATION event);

sa_reason: the reason for the provider-initiated abort (codes are listed in Table 3.4), meaningless if the abort is user-initiated; and,

sa_data/sa_cc: any abort data (and the length of that data) from the peer (if sa_peer is set) or a diagnostic string from the provider (if sa_peer is not set).

NOTE:	Although both [ISO87b] and [CCI84a] both require a maximum length of 9 octets for a user-initiated abort, the current implementation permits up to 512 octets to be used. Without this freedom, higher-layer protocols which use presentation encoding mechanisms would be unable to successfully use the session abort facility.

After examining the information returned by SInit on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the connection. For either response, the responder should use the SConnResponse routine (which corresponds to the S-CONNECT.RESPONSE action).

```
    int     SConnResponse (sd, ref, called, result, requirements,
                    settings, isn, data, cc, si)
    int     sd;
    struct SSAPref *ref;
    struct SSAPaddr *called;
    int     result,
```

	Provider-initiated Abort (fatal)
`SC_SSAPID`	SSAP identifier unknown
`SC_SSUSER`	SS-user not attached to SSAP
`SC_CONGEST`	Congestion at SSAP
`SC_VERSION`	Proposed protocol versions supported
`SC_ADDRESS`	Address unknown
`SC_REFUSED`	Connect request refused on this network connection
`SC_TRANSPORT`	Transport disconnect
`SC_ABORT`	Provider-initiated abort
`SC_PROTOCOL`	Protocol error
	User-initiated Abort (fatal)
`SC_NOTSPECIFIED`	Reason not specifed
`SC_CONGESTION`	Temporary congestion
`SC_REJECTED`	Rejected
	Interface Errors (non-fatal)
`SC_PARAMETER`	Invalid parameter
`SC_OPERATION`	Invalid operation
`SC_TIMER`	Timer expired
`SC_WAITING`	Indications waiting

Table 3.4: SSAP Failure Codes

```
        requirements,
        settings,
        cc;
long isn;
char   *data;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`ref`: the connection identifier used by the responder (consult page 52 for a description of the `SSAPref` structure);

`called`: the SSAP address of the responder (defaulting to the called address, if not present);

`result`: the acceptance indicator (either `SC_ACCEPT` if the connection is to be accepted, or one of the user-initiated abort error codes listed in Table 3.4 on page 55);

`requirements`: the responder's session requirements (this may not include any requirements not listed in the initiator's session requirements);

`settings`: the initial token settings (for each token, if the initiator specified `ST_CALL_VALUE`, then the responder should specify either `ST_INIT_VALUE` or `ST_RESP_VALUE`; instead, if the initiator specified `ST_INIT_VALUE`, and the responder wants the token, it can specify the value `ST_RSVD_VALUE`. This is interpreted by the service provider as a "tokens please" request);

`isn`: the initial serial-number;

`data/cc`: any initial data (and the length of that data, which may not exceed `SC_SIZE` octets); and,

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

If the call to SConnResponse is successful, and if the result parameter is set to SC_ACCEPT, then connection establishment has completed and the users of the session service now operate as symmetric peers. If the call is successful, but the result parameter is not SC_ACCEPT, then the connection has been rejected and the responder may exit. Otherwise, if the call fails and the reason is not an interface error (see Table 3.4 on page 55), then the connection is closed.

Note that when the responder rejects the connection, it need not supply meaningful values for the the requirements, settings, and isn parameters. The data/cc parameters are also optional, but it is recommended that the responder return some diagnostic information.

3.3.3 Client Initialization

A program wishing to connect to another user of session services calls the SConnRequest routine, which corresponds to the S-CONNECT.REQUEST action.

```
int     SConnRequest (ref, calling, called, requirements,
                settings, isn, data, cc, qos, sc, si)
struct SSAPref *ref;
struct SSAPaddr *calling,
                *called;
int     requirements,
        settings,
        cc;
long isn;
char   *data;
struct QOStype *qos;
struct SSAPconnect *sc;
struct SSAPindication *si;
```

The parameters to this procedure are:

ref: the connection identifier used by the initiator (consult page 52 for a description of the SSAPref structure);

calling: the SSAP address of the responder;

called: the SSAP address of the initiator;

`requirements`: the session requirements;

`settings`: the initial token settings;

`isn`: the initial serial-number;

`data/cc`: any initial data (and the length of that data, which may not exceed `SS_SIZE` octets);

`qos`: the quality of service on the connection (see Section 4.6.2);

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

`si`: a pointer to a `SSAPindication` structure, which is updated only if either:

- the call fails; or,
- the call succeeds, but the value of the `sc_result` element of the `sc` parameter is not `SC_ACCEPT` (see below).

If the call to `SConnRequest` is successful (a successful return corresponds to a S-CONNECT.CONFIRMATION event), then information is returned in the `sc` parameter, which is a pointer to a `SSAPconnect` structure.

```
struct SSAPconnect {
    int     sc_sd;

    struct SSAPref  sc_connect;

    struct SSAPaddr sc_responding;

    int     sc_result;

    int     sc_requirements;
    int     sc_settings;
    int     sc_please;
    lnog    sc_isn;

    int     sc_ssdusize;
```

```
    struct QOStype sc_qos;

#define SC_SIZE         512
    int     sc_cc;
    char    sc_data[SC_SIZE];
};
```

The elements of this structure are:

`sc_sd`: the session-descriptor to be used to reference this connection;

`sc_connect`: the connection identifier used by the responder (consult page 52 for a description of the `SSAPref` structure);

`sc_responding`: the responding peer's address (which is the same as the `called` parameter given to `SConnRequest`);

`sc_result`: the connection response;

`sc_requirements`: the (negotiated) session requirements;

`sc_settings`: the (negotiated) initial token settings;

`sc_please`: the tokens which the responder wants to own (if any);

`sc_isn`: the (negotiated) initial serial-number;

`sc_ssdusize`: the largest atomic SSDU size that can be used on the connection (see the note on page 45);

`sc_qos`: the quality of service on the connection (see Section 4.6.2); and,

`sc_data/sc_cc`: any initial data (and the length of that data).

If the call to `SConnRequest` is successful, and the `sc_result` element is set to `SC_ACCEPT`, then connection establishment has completed and the users of the session service now operate as symmetric peers. If the call is successful, but the `sc_result` element is not `SC_ACCEPT`, then the connection has been rejected; consult Table 3.4 to determine the reason (further information can be found in the `si` parameter). Otherwise, if the call fails then the connection is not established and the `SSAPabort` structure of the `SSAPindication` discriminated union has been updated.

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

```
int     SAsynConnRequest (ref, calling, called,
                requirements, settings, isn, data, cc,
                qos, sc, si, async)
struct SSAPref *ref;
struct SSAPaddr *calling,
                *called;
int     requirements,
        settings,
        cc,
        async;
long isn;
char   *data;
struct QOStype *qos;
struct SSAPconnect *sc;
struct SSAPindication *si;
```

The additional parameter to this procedure is:

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

If the `async` parameter is non-zero, then `SAsynConnRequest` returns one of four values: `NOTOK`, which indicates that the connection request failed; `DONE`, which indicates that the connection 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 `SConnRequest` are employed (i.e., a `NOTOK` return from `SAsynConnRequest` is equivalent to a `NOTOK` return from `SConnRequest`, and, a `DONE` return from `SAsynConnRequest` is equivalent to a `OK` return from `SConnRequest`). In the final case, when either `CONNECTING_1` or `CONNECTING_2` is returned, only the `sc_sd` element of the `sc` parameter has been updated;

it reflects the session-descriptor to be used to reference this connection. Note that the `data` parameter is still being referenced by *libssap*(3n) and should not be tampered with until the connection attempt has been completed.

To determine when the connection attempt has been completed, the routine `xselect` (consult Section 2.4 of *Volume One*) should be used after calling `SSelectMask`. In order to determine if the connection attempt is successful, the `SAsynRetryRequest` routine is called:

```
int     SAsynRetryRequest (sd, sc, si)
int     sd;
struct SSAPconnect *sc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

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

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

Again, one of three values are returned: `NOTOK`, which indicates that the connection request failed; `DONE`, which indicates that the connection 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 for information on how to make efficient use of the asynchronous connection facility.

3.4 Data Transfer

Once the connection has been established, a session-descriptor is used to reference the connection. This is usually the first parameter given to any of the remaining routines in the *libssap*(3n) library. Further, the last parameter is usually a pointer to a `SSAPindication` structure (as described on page 53). If a call to one of these routines fails, then the structure is updated. Consult the `SSAPabort` element of the `SSAPindication` structure. If the `sa_reason` element of the `SSAPabort` structure is associated with a fatal error, then the connection is closed. That is, a S-P-ABORT.INDICATION event has occurred. The SC_FATAL macro can be used to determine this.

```
int     SC_FATAL (r)
int     r;
```

The most common interface error to occur is SC_OPERATION which usually indicates that either the user is lacking ownership of a session token to perform an operation, or that a session requirement required by the operation was not negotiated during connection establishment. For protocol purists, the SC_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     SC_OFFICIAL (r)
int     r;
```

3.4.1 Sending Data

There are six routines which may be used to send data. A call to the SDataRequest routine is equivalent to a S-DATA.REQUEST action on the part of the user.

```
int     SDataRequest (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

sd: the session-descriptor;

data/cc: the data to be written (and the length of that data); and,

si: a pointer to a SSAPindication structure, which is updated only if the call fails.

If the call to SDataRequest is successful, then the data has been queued for sending to the peer. Otherwise the SSAPabort structure contained in the SSAPindication parameter si contains the reason for failure.

A call to the SExpdRequest routine is equivalent to a S-EXPEDITED-DATA.REQUEST action on the part of the user.

```
int     SExpdRequest (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`data/cc`: the data to be written (and the length of that data, which may not exceed `SX_SIZE` octets); and,

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

If the call to `SExpdRequest` is successful, then the data has been queued for expedited sending. Otherwise the `SSAPabort` element of the `si` parameter contains the reason for failure.

A call to the `STypedRequest` routine is equivalent to a S-TYPED-DATA.REQUEST action on the part of the user.

```
int     STypedRequest (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`data/cc`: the data to be written (and the length of that data); and,

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

If the call to `STypedRequest` is successful, then the data has been queued for sending to the peer. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

A call to the `SCapdRequest` routine is equivalent to a S-CAPABILITY-DATA.REQUEST action on the part of the user.

```
int     SCapdRequest (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

sd: the session-descriptor;

data/cc: the data to be written (and the length of that data, which may not exceed SX_CDSIZE octets); and,

si: a pointer to a SSAPindication structure, which is updated only if the call fails.

If the call to SCapdRequest is successful, then the data has been queued for sending. When the S-CAPABILITY-DATA.CONFIRMATION event is received, the data has been successfully received. Otherwise the SSAPabort structure contained in the SSAPindication parameter si contains the reason for failure.

Upon receiving a S-CAPABILITY-DATA.INDICATION event, the user is required to generate a S-CAPABILITY-DATA.RESPONSE action using the SCapdResponse routine.

```
int     SCapdResponse (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

sd: the session-descriptor;

data/cc: the data to be written (and the length of that data, which may not exceed SX_CDASIZE octets); and,

si: a pointer to a SSAPindication structure, which is updated only if the call fails.

If the call to `SCapdResponse` is successful, then the data has been queued for sending to the peer. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

The `SWriteRequest` routine is similar in nature to the `SDataRequest` and `STypedRequest` routines, but uses a different set of parameters. The invocation is:

```
int     SWriteRequest (sd, typed, uv, si)
int     sd;
int     typed;
struct udvec *uv;
int     cc;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`typed`: whether this is typed-data or not;

`uv`: the data to be written, described in a null-terimated array of scatter/gather elements; and,

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

If the call to `SWriteRequest` is successful, then the data has been queued for sending to the peer. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

3.4.2 Receiving Data

There is one routine used to read data, `SReadRequest`, a call to which is equivalent to waiting for an event (usually an incoming data event) to occur.

```
int     SReadRequest (sd, sx, secs, si)
int     sd;
struct SSAPdata *sx;
int     secs;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`sx`: a pointer to the `SSAPdata` structure to be given the data;

`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,

`si`: a pointer to a `SSAPindication` structure, which is updated only if data is not read.

Unlike the other routines in the *libssap*(3n) library, the `SReadRequest` routine returns one of three values: `NOTOK` (on failure), `OK` (on reading data), or `DONE` (otherwise).

If the call to `SReadRequest` returns the manifest constant `OK`, then the data has been read into the `sx` parameter, which is a pointer to a `SSAPdata` structure.

```
struct SSAPdata {
    int     sx_type;
#define SX_NORMAL       0x00
#define SX_EXPEDITED    0x01
#define SX_TYPED        0x02
#define SX_CAPDIND      0x03
#define SX_CAPDCNF      0x04

    int     sx_cc;
    struct qbuf sx_qbuf;
};
```

The elements of a `SSAPdata` structure are:

`sx_type`: indicates how the data was received:

Value	Event
SX_NORMAL	S-DATA.INDICATION
SX_EXPEDITED	S-EXPEDITED-DATA.INDICATION
SX_TYPED	S-TYPED-DATA.INDICATION
SX_CAPDIND	S-CAPABILITY-DATA.INDICATION
SX_CAPDCNF	S-CAPABILITY-DATA.CONFIRMATION

`sx_cc`: the total number of octets that was read; and,

`sx_qbuf`: the data that was read in a buffer-queue form (see page 119 for a description of this structure).

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

```
void    SXFREE (sx)
struct SSAPdata *sx;
```

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

NOTE:	Because the `SSAPdata` structure contains a `qbuf` element, care must be taken in initializing and copying variables of this type. The routines in *libssap*(3n) library will correctly initialize these structures when given as parameters. But, users who otherwise manipulate these structures should take great care.

Otherwise if the call to `SReadRequest` returns the manifest constant `NOTOK`, then the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

If the call to `SReadRequest` returns the manifest constant `DONE`, then some event other than data arrival has occurred. This event is encoded in the `si` parameter, depending on the value of the `si_type` element. When `SReadRequest` returns `DONE`, the `si_type` element may be set to one of five values:

Value	Event
`SI_TOKEN`	Token management
`SI_SYNC`	Synchronization management
`SI_ACTIVITY`	Activity management
`SI_REPORT`	Exception reporting
`SI_FINISH`	Connection release

Token Indications

When an event associated with token management occurs, the `si_type` field of the `si` parameter contains the value `SI_TOKEN`, and a `SSAPtoken` structure is contained inside the `si` parameter.

```
struct SSAPtoken {
    int     st_type;
#define ST_GIVE         0x00
#define ST_PLEASE       0x01
#define ST_CONTROL      0x02

    u_char  st_tokens;
    u_char  st_owned;

#define ST_SIZE         512
    int     st_cc;
    char    st_data[ST_SIZE];
};
```

The elements of a `SSAPtoken` structure are:

`st_type`: defines the token management indication which occurred:

Value	Event
ST_GIVE	S-TOKEN-GIVE.INDICATION
ST_PLEASE	S-TOKEN-PLEASE.INDICATION
ST_CONTROL	S-GIVE-CONTROL.INDICATION

`st_tokens`: the session tokens requested or given;

`st_owned`: all of the session tokens currently owned by the user; and,

`st_base/st_cc`: associated data (and the length of that data) if tokens are requested.

It is entirely at the discretion of the user what actions are to be taken when an indication event associated with token management occurs.

Synchronization Indications

When an event associated with synchronization occurs, the `si_type` field of the `si` parameter contains the value `SI_SYNC`, and a `SSAPsync` structure is contained inside the `si` parameter.

```
struct SSAPsync {
    int     sn_type;
#define SN_MAJORIND     0x00
#define SN_MAJORCNF     0x01
#define SN_MINORIND     0x02
#define SN_MINORCNF     0x03
#define SN_RESETIND     0x04
#define SN_RESETCNF     0x05

    int     sn_options;
#define SYNC_CONFIRM    1
#define SYNC_NOCONFIRM  0

#define SYNC_RESTART    0
#define SYNC_ABANDON    1
#define SYNC_SET        2

    long    sn_ssn;
#define SERIAL_NONE     (-1L)
#define SERIAL_MIN      000000L
#define SERIAL_MAX      999998L

    int     sn_settings;

#define SN_SIZE         512
    int     sn_cc;
    char    sn_data[SN_SIZE];
};
```

The elements of a `SSAPsync` structure are:

`sn_type`: defines the synchronization management indication which occurred:

Value	Event
SN_MAJORIND	S-MAJOR-SYNC.INDICATION
SN_MAJORCNF	S-MAJOR-SYNC.CONFIRMATION
SN_MINORIND	S-MINOR-SYNC.INDICATION
SN_MINORCNF	S-MINOR-SYNC.CONFIRMATION
SN_RESETIND	S-RESYNCHRONIZE.INDICATION
SN_RESETCNF	S-RESYNCHRONIZE.CONFIRMATION

`sn_options`: various modifiers of the indication. For the minorsync indication (as described in Section 3.4.4 on page 76):

Value	Modifier
SYNC_CONFIRM	peer wants explicit confirmation
SYNC_NOCONFIRM	peer doesn't want explicit confirmation

For the resync indication (also described in Section 3.4.4):

Value	Modifier
SYNC_RESTART	a "restart" resynchronization is requested
SYNC_ABANDON	a "abandon" resynchronization is requested
SYNC_SET	a "set" resynchronization is requested

`sn_ssn`: the serial-number associated with this synchronization management event;

`sn_settings`: for resync events, the proposed (resync indication) or new (resync confirmation) token settings; and,

`sn_data/sn_cc`: associated data (and the length of that data).

Note that for minorsync events, the user is not obligated to confirm the synchronization point even if the originator requested it.

Activity Indications

When an event associated with activity management occurs, the `si_type` field of the `si` parameter contains the value `SI_ACTIVITY`, and the `si` parameter contains a `SSAPactivity` structure.

```
struct SSAPactivity {
    int     sv_type;
#define SV_START        0x00
#define SV_RESUME       0x01
#define SV_INTRIND      0x02
#define SV_INTRCNF      0x03
#define SV_DISCIND      0x04
#define SV_DISCCNF      0x05
#define SV_ENDIND       0x06
#define SV_ENDCNF       0x07

    struct SSAPactid sv_id;

    struct SSAPactid sv_oid;
    struct SSAPref   sv_connect;

    long    sv_ssn;

    int     sv_reason;

#define SV_SIZE         512
    int     sv_cc;
    char    sv_data[SV_SIZE];
};
```

The elements of a `SSAPactivity` structure are:

sv_type: defines the activity management indication which occurred:

Value	Event
SV_START	S-ACTIVITY-START.INDICATION
SV_RESUME	S-ACTIVITY-RESUME.INDICATION
SV_INTRIND	S-ACTIVITY-INTERRUPT.INDICATION
SV_INTRCNF	S-ACTIVITY-INTERRUPT.CONFIRMATION
SV_DISCIND	S-ACTIVITY-DISCARD.INDICATION
SV_DISCCNF	S-ACTIVITY-DISCARD.CONFIRMATION
SV_ENDIND	S-ACTIVITY-END.INDICATION
SV_ENDCNF	S-ACTIVITY-END.CONFIRMATION

sv_id: the activity identifier for an activity start or resume indication;

sv_oid: the previous activity identifier for an activity resume indication (see page 82);

sv_connect: the previous connection identifier for an activity resume indication;

sv_ssn: the serial-number for an activity resume or end indication;

sv_reason: the reason for an activity interrupt or discard indication (codes are listed in Table 3.5); and,

sv_data/sv_cc: associated data (and the length of that data).

Report Indications

When an event associated with exception reporting occurs, the si_type field of the si parameter contains the value SI_REPORT, and a SSAPreport structure is contained inside the si parameter.

```
struct SSAPreport {
    int     sp_peer;

    int     sp_reason;

#define SP_SIZE        512
```

	Provider-initiated Report
SP_NOREASON	No specific reason stated
SP_PROTOCOL	SS-provider protocol error
	User-initiated Report
SP_NOREASON	No specific reason stated
SP_JEOPARDY	User receiving ability jeopardized
SP_SEQUENCE	User sequence error
SP_LOCAL	Local SS-user error
SP_PROCEDURAL	Unrecoverable procedural error
SP_DEMAND	Demand data token

Table 3.5: SSAP Exception Codes

```
        int     sp_cc;
        char    sp_data[SP_SIZE];
    };
```

The elements of a `SSAPreport` structure are:

`sp_peer`: if set, indicates that a user-initiated report occurred (a S-U-EXCEPTION-REPORT.INDICATION event); if not set, indicates that a provider-initiated report occurred (a S-P-EXCEPTION-REPORT.INDICATION event);

`sp_reason`: the reason for the report (codes are listed in Table 3.5 on page 73); and,

`sp_data/sp_cc`: any report data (and the length of that data) from the peer; meaningless if the report is provider-initiated.

Finish Indication

When a S-RELEASE.INDICATION event occurs, the `si_type` field of the `si` parameter contains the value `SI_FINISH`, and a `SSAPfinish` structure is contained inside the `si` parameter.

```
struct SSAPfinish {
#define SF_SIZE         512
    int     sf_cc;
    char    sf_data[SF_SIZE];
};
```

The elements of a `SSAPfinish` structure are:

`sf_data/sf_cc`: any final data (and the length of that data).

3.4.3 Token Management

The fundamental aspect of token management deals with transferring ownership of the tokens.

Sending Tokens

To transfer ownership of one or more session tokens to the remote user, the `SGTokenRequest` routine is called (which corresponds to the S-TOKEN-GIVE.REQUEST action).

```
int     SGTokenRequest (sd, tokens, si)
int     sd;
int     tokens;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`tokens`: the tokens to be transferred; and,

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

If the call to `SGTokenRequest` is successful, then ownership of the tokens has been transferred to the remote user.

If activity management has been selected, then the ownership of all tokens can be transferred using the `SGControlRequest` routine (which corresponds to the S-CONTROL-GIVE.REQUEST action).

```
int     SGControlRequest (sd, si)
int     sd;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor; and,

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

If the call to `SGControlRequest` is successful, then ownership of all available tokens has been transferred to the remote user. Until this transfer of ownership is acknowledged, other token management functions will (non-fatally) fail.

Requesting Tokens

To request ownership of one or more session tokens, the `SPTokenRequest` routine is called (which corresponds to the S-TOKEN-PLEASE.REQUEST action).

```
int     SPTokenRequest (sd, tokens, data, cc, si)
int     sd;
int     tokens,
        cc;
char   *data;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`tokens`: the tokens to requested;

`data/cc`: any additional data (and the length of that data, which may not exceed `ST_SIZE` octets); and,

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

If the call to `SPTokenRequest` is successful, then the remote user has been notified of the request; however, the ownership of the tokens is not actually transferred until the session provider notifies the user with a S-TOKEN-GIVE-INDICATION event, which typically occurs on the next call to `SReadRequest`.

3.4.4 Synchronization Management

There are three types of synchronization services: majorsyncs, minorsyncs, and resyncs.

Major Synchronization

To indicate a major synchronization point, the `SMajSyncRequest` routine is used (which corresponds to the S-MAJOR-SYNC.REQUEST action).

```
int     SMajSyncRequest (sd, ssn, data, cc, si)
int     sd;
long   *ssn;
char   *data;
int     cc;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`ssn`: a pointer to a long integer which, on a successful return, will be updated to reflect the current serial-number ($V(M) - 1$);

`data/cc`: any additional data (and the length of that data, which may not exceed `SN_SIZE` octets); and,

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

If the call to `SMajSyncRequest` is successful, then the major synchronization has been queued for the remote user. When the S-MAJOR-SYNC.CONFIRMATION event is received, the major synchronization is complete. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Upon receiving a S-MAJOR-SYNC.INDICATION event, the user is required to generate a S-MAJOR-SYNC.RESPONSE action by calling the `SMajSyncResponse` routine.

```
int     SMajSyncResponse (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`data/cc`: any additional data (and the length of that data, which may not exceed `SN_SIZE` octets); and,

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

If the call to `SMajSyncResponse` is successful, then the major synchronization has been completed. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Minor Synchronization

To indicate a minor synchronization point, the `SMinSyncRequest` routine is used (which corresponds to the S-MINOR-SYNC.REQUEST action).

```
int     SMinSyncRequest (sd, type, ssn, data, cc, si)
int     sd;
int     type;
long   *ssn;
char   *data;
int     cc;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`type`: the type of minor synchronization requested, one of:

Value	Type
SYNC_CONFIRM	explicit confirmation requested
SYNC_NOCONFIRM	no confirmation requested

`ssn`: a pointer to a long integer which, on a successful return, will be updated to reflect the current serial-number ($V(M)-1$);

`data/cc`: any additional data (and the length of that data, which may not exceed `SN_SIZE` octets); and,

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

If the call to `SMinSyncRequest` is successful, then the minor synchronization has been queued for the remote user. If a S-MINOR-SYNC.CONFIRMATION event is received, the minor synchronization is complete. However, the remote user is under no obligation to acknowledge the minorsync. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Upon receiving a S-MINOR-SYNC.INDICATION event, the user may optionally use the `SMinSyncResponse` routine to generate a S-MINOR-SYNC.RESPONSE action.

```
int     SMinSyncResponse (sd, ssn, data, cc, si)
int     sd;
long    ssn;
char   *data;
int     cc;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`ssn`: the highest serial-number being confirmed;

`data/cc`: any additional data (and the length of that data, which may not exceed `SN_SIZE` octets); and,

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

If the call to `SMinSyncResponse` is successful, then the minor synchronization has been completed. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

ReSynchronization

To resynchronize the connection to a known state, the `SReSyncRequest` is used (which corresponds to the S-RESYNCHRONIZE.REQUEST action).

```
int     SReSyncRequest (sd, type, ssn, settings, data, cc, si)
int     sd;
int     type,
        settings;
long    ssn;
char   *data;
int     cc;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`type`: the type of resynchronization requested (either `SYNC_RESTART`, `SYNC_ABANDON`, or `SYNC_SET`);

`ssn`: the serial-number to resynchronize to;

`settings`: the new token settings;

`data/cc`: any additional data (and the length of that data, which may not exceed `SN_SIZE` octets); and,

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

If the call to `SReSyncRequest` is successful, then the resynchronization has been queued for the remote user. When the S-RESYNCHRONIZE.CONFIRMATION event is received, the resynchronization is complete. Otherwise the SSAPabort structure contained in the SSAPindication parameter si contains the reason for failure.

Upon receiving a S-RESYNCHRONIZE.INDICATION event, the user is required to generate a S-RESYNCHRONIZE.RESPONSE action using using the SReSyncResponse routine.[2]

```
int     SReSyncResponse (sd, ssn, settings, data, cc, si)
int     sd;
int     settings;
long    ssn;
char   *data;
int     cc;
struct SSAPindication *si;
```

The parameters to this procedure are:

sd: the session-descriptor;

ssn: the serial-number to resynchronize to;

settings: the new token settings;

data/cc: any additional data (and the length of that data, which may not exceed SN_SIZE octets); and,

si: a pointer to a SSAPindication structure, which is updated only if the call fails.

3.4.5 Activity Management

There are several types of activity management services: activity start and resume, activity interrupt and discard, and activity end.

Activity Start/Resume

To initiate a new activity, the SActStartRequest routine is used (which corresponds to the S-ACTIVITY-START.REQUEST action).

[2] Actually, the user has other choices by using the rules of contention resolution. Consult the ISO session service specification for the gruesome details.

```
int     SActStartRequest (sd, id, data, cc, si)
int     sd;
struct SSAPactid *id;
char   *data;
int     cc;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`id`: the activity-identifier;

`data/cc`: any additional data (and the length of that data, which may not exceed `SV_SIZE` octets); and,

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

The `id` parameter is a pointer to a `SSAPactid` structure, which is passed transparently by the session service.

```
struct SSAPactid {
#define SID_DATA_SIZE   6
    u_char  sd_len;
    char    sd_data[SID_DATA_SIZE];
};
```

The elements of this structure are:

`sd_data/sd_len`: the data (and length of that data, which may not exceed `SID_DATA_SIZE` octets);

If the call to the `SActStartRequest` routine is successful, then the activity is started. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

To resume a previously interrupted activity, the `SActResumeRequest` routine is used (which corresponds to the S-ACTIVITY-RESUME.REQUEST action).

```
int     SActResumeRequest (sd, id, oid, ssn, ref, data, cc,
                si)
int     sd;
struct SSAPactid *id,
                 *oid;
long    ssn;
struct SSAPref *ref;
char   *data;
int     cc;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`id`: the activity-identifier (consult page 81 for a description of the `SSAPactid` structure);

`oid`: the previous activity-identifier (again, consult page 81);

`ssn`: the serial-number to resume the activity at;

`ref`: the previous connection identifier;

`data/cc`: any additional data (and the length of that data, which may not exceed `SV_SIZE` octets); and,

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

The `ref` parameter is a pointer to a `SSAPref` structure. Note that unlike the connection identifiers used during connection establishment (as described on page 52), there are four fields:

Field	**Contents**	**Length**
calling SSAP user reference	`sr_calling`	`sr_calling_len`
called SSAP user reference	`sr_called`	`sr_called_len`
common reference	`sr_cdata`	`sr_clen`
additional reference	`sr_adata`	`sr_alen`

If the call to the `SActResumeRequest` routine is successful, then the activity is resumed. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Activity Interrupt/Discard

To interrupt an activity in progress, the `SActIntrRequest` routine is used (which corresponds to the S-ACTIVITY-INTERRUPT.REQUEST action).

```
int     SActIntrRequest (sd, reason, si)
int     sd;
int     reason;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`reason`: the reason for the interrupt (codes are listed in Table 3.5); and,

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

If the call to `SActIntrRequest` is successful, then the activity interrupt has been queued for the remote user. When the S-ACTIVITY-INTERRUPT.CONFIRMATION event is received, the activity interrupt is complete. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Upon receiving a S-ACTIVITY-INTERRUPT.INDICATION event, the user is required to generate a S-ACTIVITY-INTERRUPT.RESPONSE action using the `SActIntrResponse` routine.

```
int     SActIntrResponse (sd, si)
int     sd;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor; and,

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

If the call to `SActIntrResponse` is successful, then the activity interrupt has been completed. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

To discard an activity in progress, the `SActDiscRequest` routine is used (which corresponds to the S-ACTIVITY-DISCARD.REQUEST action).

```
int     SActDiscRequest (sd, reason, si)
int     sd;
int     reason;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`reason`: the reason for the discard (codes are listed in Table 3.5); and,

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

If the call to `SActDiscRequest` is successful, then the activity discard has been queued for the remote user. When the S-ACTIVITY-DISCARD.CONFIRMATION event is received, the activity discard is complete. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Upon receiving a S-ACTIVITY-DISCARD.INDICATION event, the user is required to generate a S-ACTIVITY-DISCARD.RESPONSE action using the `SActDiscResponse` routine.

```
int     SActDiscResponse (sd, si)
int     sd;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor; and,

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

If the call to `SActDiscResponse` is successful, then the activity discard has been completed. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Activity End

To end an activity in progress, the `SActEndRequest` routine is used (which corresponds to the S-ACTIVITY-END.REQUEST action).

```
int     SActEndRequest (sd, ssn, data, cc, si)
int     sd;
long   *ssn;
char   *data;
int     cc;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`ssn`: a pointer to a long integer which, on a successful return, will be updated to reflect the current serial-number ($V(M) - 1$);

`data/cc`: any additional data (and the length of that data, which may not exceed `SV_SIZE` octets); and,

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

If the call to `SActEndRequest` is successful, then the activity end has been queued for the remote user. When the S-ACTIVITY-END.CONFIRMATION event is received, the activity end is complete. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Upon receiving a S-ACTIVITY-END.INDICATION event, the user is required to call the `SActEndResponse` routine to generate a S-ACTIVITY-END.RESPONSE action.

```
int     SActEndResponse (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`data/cc`: any additional data (and the length of that data, which may not exceed `SV_SIZE` octets); and,

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

If the call to `SActEndResponse` is successful, then the activity end has been completed. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

3.4.6 Exception Reporting

To report an exception and place the connection in a special error-recovery state, the `SUReportRequest` routine is called (which corresponds to the S-U-EXCEPTION-REPORT.REQUEST action).

```
int SUReportRequest (sd, reason, data, cc, si)
int sd;
int     reason;
char   *data;
int cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`reason`: the reason for the report (codes are listed in Table 3.5);

`data/cc`: any report data (and the length of that data, which may not exceed `SP_SIZE` octets); and,

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

If the call to `SUReportRequest` is successful, then the connection is placed in an error state, and any data queued for the connection may be lost until recovery is complete. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Typically, error recovery can be achieved by giving away the data token or by aborting the connection (discussed next). Error recovery is discussed in greater detail in the ISO session service specification.

3.4.7 User-initiated Aborts

To unilaterally initiate an abort of the connection, the `SUAbortRequest` routine is called (which corresponds to the S-U-ABORT.REQUEST action).

```
int SUAbortRequest (sd, data, cc, si)
int sd;
char   *data;
int cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`data/cc`: any abort data (and the length of that data, which may not exceed `SA_SIZE` octets); and,

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

If the call to `SUAbortRequest` is successful, then the connection is immediately closed, and any data queued for the connection may be lost.

3.4.8 Asynchronous Event Handling

The data transfer events discussed thus far have been synchronous in nature. Some users of the session service may wish an asynchronous interface. The `SSetIndications` routine is used to change the service associated with a session-descriptor to or from an asynchronous interface.

```
int     SSetIndications (sd, data, tokens, sync, activity,
                report, finish, abort, si)
int     sd;
int   (*data) (),
      (*tokens) (),
```

```
        (*sync) (),
        (*activity) (),
        (*report) (),
        (*finish) (),
        (*abort) ();
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`data`: the address of an event-handler routine to be invoked when data has arrived;

`tokens`: the address of an event-handler routine to be invoked when a token management event occurs;

`sync`: the address of an event-handler routine to be invoked when a synchronization management event occurs;

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

`report`: the address of an event-handler routine to be invoked when an exception report event occurs;

`finish`: the address of an event-handler routine to be invoked when the connection is ready to be released;

`abort`: the address of an event-handler routine to be invoked when a user-initiated abort (a S-U-ABORT.INDICATION event occurs) or a provider-initiated abort (a S-P-ABORT.INDICATION event occurs); and,

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

If the service is to be made asynchronous, then all event handlers are specified, otherwise, if the service is to be made synchronous, then no event handlers should be specified (use the manifest constant `NULLIFP`). The most likely reason for the call failing is SC_WAITING, which indicates that an event is waiting for the user.

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.

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

```
(*data) (sd, sx);
int     sd;
struct SSAPdata *sx;
```

The parameters are:

sd: the session-descriptor; and,

sx: a pointer to the SSAPdata structure containing the data.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released with the SXFREE macro (described on page 67) when no longer needed.

When an event associated with token management arrives the event-handler routine is invoked with two parameters:

```
(*tokens) (sd, st);
int       sd;
struct SSAPtoken *st;
```

The parameters are:

sd: the session-descriptor; and,

st: a pointer to the SSAPtoken structure containing the token management information.

When an event associated with synchronization management arrives the event-handler routine is invoked with two parameters:

```
(*sync) (sd, sn);
int       sd;
struct SSAPsync *sn;
```

The parameters are:

`sd`: the session-descriptor; and,

`sn`: a pointer to the `SSAPsync` structure containing the synchronization management information.

When an event associated with activity management arrives the event-handler routine is invoked with two parameters:

```
(*activity) (sd, sv);
int        sd;
struct SSAPactivity *sv;
```

The parameters are:

`sd`: the session-descriptor; and,

`sv`: a pointer to the `SSAPactivity` structure containing the activity management information.

When an event associated with exception reporting arrives the event-handler routine is invoked with two parameters:

```
(*report) (sd, sp);
int        sd;
struct SSAPreport *sp;
```

The parameters are:

`sd`: the session-descriptor; and,

`sp`: a pointer to the `SSAPreport` structure containing the exception report information.

When an event associated with connection termination arrives the event-handler routine is invoked with two parameters:

```
(*finish) (sd, sf);
int        sd;
struct SSAPfinish *sf;
```

The parameters are:

`sd`: the session-descriptor; and,

`sf`: a pointer to the `SSAPfinish` structure containing information concerning the request to terminate the connection.

When an event associated with a user- or provider-initiated abort occurs, the event-handler is invoked with two parameters:

```
(*abort) (sd, sa);
int      sd;
struct SSAPabort *sa;
```

The parameters are:

`sd`: the session-descriptor; and,

`sa`: a pointer to the `SSAPabort` structure indicating why the connection was aborted.

Note that the session-descriptor is no longer valid at the instant the call is made.

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

3.4.9 Synchronous Event Multiplexing

A user of the session service may wish to manage multiple session-descriptors simultaneously; the routine `SSelectMask` is provided for this purpose. This routine updates a file-descriptor mask and associated counter for use with `xselect`.

```
int     SSelectMask (sd, mask, nfds, si)
int     sd;
fd_set *mask,
int    *nfds;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

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

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

`si`: a pointer to a `SSAPindication` 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 `SC_WAITING`, which indicates that an event is waiting for the user.

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

3.5 Connection Release

The `SRelRequest` routine is used to request the release a connection, and corresponds to a S-RELEASE.REQUEST action. The SSAP attempts to gracefully drain any queued data before closing the connection.

```
int     SRelRequest (sd, data, cc, secs, sr, si)
int     sd;
char   *data;
int     cc;
int     secs;
struct SSAPrelease  *sr;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`data/cc`: any final data (and the length of that data, which may not exceed `SF_SIZE` octets);

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

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

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

If the call to `SRelRequest` is successful, then this corresponds to a S-RELEASE.CONFIRMATION event, and it returns information in the `sr` parameter, which is a pointer to a `SSAPrelease` structure.

```
struct SSAPrelease {
    int     sr_affirmative;

#define SR_SIZE         512
    int     sr_cc;
    char    sr_data[SR_SIZE];
};
```

The elements of this structure are:

`sr_affirmative`: the acceptance indicator; and,

`sr_data/sr_cc`: any final data (and the length of that data).

If the call to `SRelRequest` is successful, and the `sr_affirmative` element is set, then the connection has been closed. If the call is successful, but the `sr_affirmative` element is not set (i.e., zero), then the request to close the connection has been rejected by the remote user, and the connection is still open. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` 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 `sa_reason` element is `SC_TIMER`. The user can then call the routine `SRelRetryRequest` to continue waiting for a response:

```
int     SRelRetryRequest (sd, secs, sr, si)
int     sd;
```

```
int     secs;
struct SSAPrelease  *sr;
struct SSAPindication  *si;
```

The parameters to this procedure are:

sd: the session-descriptor;

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

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

si: a pointer to a SSAPindication structure, which is updated only if the call fails.

If the call to SRelRetryRequest is successful, and the sr_affirmative element is set, then the connection has been closed. If the call is successful, but the sr_affirmative element is not set (i.e., zero), then the request to close the connection has been rejected by the remote user, and the connection is still open. Otherwise the SSAPabort structure contained in the SSAPindication parameter si contains the reason for failure. As expected, the value SC_TIMER indicates that no response was received within the time given by the secs parameter.

Upon receiving a S-RELEASE.INDICATION event, the user is required to generate a S-RELEASE.RESPONSE action using the SRelResponse routine.

```
int     SRelResponse (sd, status, data, cc, si)
int     sd;
int     status,
        cc;
char   *data;
struct SSAPindication  *si;
```

The parameters to this procedure are:

sd: the session-descriptor;

status: the acceptance indicator (either SC_ACCEPT if the connection is to be closed, or SC_REJECTED otherwise);

`data/cc`: any final data (and the length of that data, which may not exceed `SR_SIZE` octets); and,

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

If the call to `SRelResponse` is successful, and if the `result` parameter is set to `SC_ACCEPT`, then the connection has been closed. If the call is successful, but the `result` parameter is not `SC_ACCEPT`, then the connection still remains open. Note that in order to specify a value other tha `SC_ACCEPT` for the `result` parameter to `SRelResponse`, the release token must exist but must not be owned by the user making the call to `SRelResponse`.[3]

3.6 Restrictions on User Data

The *libssap*(3n) contains partial support for the use of unlimited user data for session services. With the exception of the S-DATA and the S-TYPED-DATA services, the initial session service limited the amount of user data that could be present. With the introduction of the unlimited user data addendum[ISO87c, ISO87d] to the session service and protocol, this restriction has been lifted.

During connection establishment, the *libssap*(3n) library will attempt to negotate the use of unlimited user data. If this negotiation fails, then session services which permit user data, other than S-DATA and S-TYPED-DATA, are limited to 512 octets of user data.[4]

If the negotation succeeds, then session services which permit user data, other than S-DATA and S-TYPED-DATA, are limited to 65400 octets of user data, with the exception of the S-CONNECT.REQUEST primitive, which is limited to 10240 octets of user data (the S-CONNECT.RESPONSE primitive is limited to 65528 octets).[5]

There is one further limitation however: although the initiator of a connection can send upto 10240 octets, due to limitations in the UNIX kernel,

[3] Gentle reader, we don't write the standards; we just try to implement them.

[4] Strictly speaking, the S-U-ABORT service permits only 9 octets of user data. This limitation is unreasonable — upto 512 octets will be permitted.

[5] Strictly speaking, the amount should be unlimited. However this full generality is not available at this time.

if the *tsapd*(8c) is dispatching based on session selector, then a responder can accept upto approximately 1536 octets. To avoid this problem, have *tsapd*(8c) dispatch based on transport selector (see Section 10.1 in *Volume One*).

3.7 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `si` parameter given to the routine. The `si_abort` element of the `SSAPindication` structure contains a `SSAPabort` structure detailing the reason for the failure. The `sa_reason` element of this latter structure can be given as a parameter to the routine `SErrString` which returns a null-terminated diagnostic string.

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

3.8 Compiling and Loading

Programs using the *libssap*(3n) library should include `<isode/ssap.h>`. The programs should also be loaded with `-lssap` and `-ltsap` (this latter library contains the routines which implement the transport services used by the session provider).

3.9 An Example

Let's consider how one might construct a source entity that resides on the SSAP. This entity will use a synchronous interface.

There are two parts to the program: initialization and data transfer; release will occur when the standard input has been exhausted. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure 3.3, the initialization steps for the source entity, including the outer *C* wrapper, is shown. First, a lookup is done in the ISO services database, and the `SSAPaddr` is initialized. The `SSAPref` is zeroed. Next, for each token associated with the session requirements, initial ownership of that token is claimed. Finally, the call to `SConnRequest` is made. If the call is successful, a check is made to see if the remote user accepted the connection. If so, the session-descriptor is captured, along with the negotiated requirements and initial token settings.

In Figure 3.4 on page 100, the data transfer loop is realized. The source entity reads a line from the standard input, and then queues that line for sending to the remote side. When an end-of-file occurs on the standard input, the source entity requests release and then gracefully terminates. Although no checking is done in this example, for the calls to `SDataRequest` and `SRelRequest`, on failure a check for the operational error `SC_OPERATION` should be made. For `SDataRequest`, this would occur when the data token was not owned by the user; for `SRelRequest`, this would occur when the release token was not owned by the user.

```
#include <stdio.h>
#include <isode/ssap.h>
#include <isode/isoservent.h>


static int requirements = SR_HALFDUPLEX | SR_NEGOTIATED;

static int owned = 0;


/* ARGSUSED */

main (argc, argv, envp)
int     argc;
char  **argv,
      **envp;
{
    int     sd,
            settings;
    char    buffer[BUFSIZ];
    register struct SSAPaddr   *sz;
    struct SSAPref  sfs;
    register struct SSAPref *sf = &sfs;
    struct SSAPconnect  scs;
    register struct SSAPconnect *sc = &scs;
    struct SSAPrelease  srs;
    register struct SSAPrelease *sr = &srs;
    struct SSAPindication   sis;
    register struct SSAPindication *si = &sis;
    register struct SSAPabort *sa = &si -> si_abort;
    register struct isoservent *is;

    if ((is = getisoserventbyname ("sink", "ssap")) == NULL)
        error ("ssap/sink:  unknown provider/entity pair");
    if (argc != 2)
        error ("usage:  source \"host\"");

...
```

Figure 3.3: Initializing the SSAP source entity

```
...

    if ((sz = is2saddr (argv[1], NULLCP, is)) == NULLSA)
            error ("address translation failed");
    (void) bzero ((char *) sf, sizeof *sf);

    settings = 0;
#define dotoken(requires,shift,bit,type) \
{ \
    if (requirements & requires) \
            settings |= ST_INIT_VALUE << shift; \
}
    dotokens ();
#undef dotoken

    if (SConnRequest (sf, NULLSA, sz, requirements, settings, SERIAL_NONE,
                NULLCP, 0, NULLQOS, sc, si) == NOTOK)
            error ("S-CONNECT.REQUEST: %s", SErrString (sa -> sa_reason));
    if (sc -> sc_result != SC_ACCEPT)
            error ("connection rejected by sink[%s]:  %s",
                        SErrString (sa -> sa_reason),
                        SErrString (sc -> sc_result));

    sd = sc -> sc_sd;
    requirements = sc -> sc_requirements;

#define dotoken(requires,shift,bit,type) \
{ \
    if (requirements & requires) \
            if ((sc -> sc_settings & (ST_MASK << shift)) == ST_INIT_VALUE) \
                owned |= bit; \
}
            dotokens ();
#undef dotoken

...
```

Figure 3.3: Initializing the SSAP source entity (continued)

```
...

    while (fgets (buffer, sizeof buffer, stdin))
            if (SDataRequest (sd, buffer, strlen (buffer) + 1, si) == NOTOK)
                error ("S-DATA.REQUEST: %s", SErrString (sa -> sa_reason));

    if (SRelRequest (sd, NULLCP, 0, NOTOK, sr, si) == NOTOK)
            error ("S-RELEASE.REQUEST: %s", SErrString (sa -> sa_reason));

    if (!sr -> sr_affirmative) {
            (void) SUAbortRequest (sd, NULLCP, 0, si);
            error ("release rejected by sink");
    }

    exit (0);
}
```

Figure 3.4: Data Transfer for the SSAP source entity

.

3.10 For Further Reading

The ISO specification for session services is defined in [ISO87e], while the complementary CCITT recommendation is defined in [CCI84b]. The corresponding protocol definitions are [ISO87b] and [CCI84a], respectively.

Chapter 4

Transport Services

At the heart of this distribution is the *libtsap*(3n) library. This library contains a set of routines which implement the transport services access point (TSAP).

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 necessarily 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 `tsap` 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 *libtsap*(3n) library are integer-valued. They return the manifest constant `OK` on success, or `NOTOK` otherwise.

4.1 Addresses

Addresses at the transport service access point are represented by the `TSAPaddr` structure.

```
struct TSAPaddr {
#define NTADDR  4
    struct NSAPaddr ta_addrs[NTADDR];
    int     ta_naddr;

#define TSSIZE  64
    int     ta_selectlen;
    char    ta_selector[TSSIZE];
};
#define NULLTA  ((struct TSAPaddr *) 0)
```

This structure contains two elements:

`ta_addrs/ta_nadr`: a list of network addresses, as described in the Section 4.6.1 on page 125;

`ta_selector/ta_selectlen`: the transport selector.

In Figure 4.1, an example of how one constructs the TSAP address for the session provider on host `RemoteHost` is presented. The routine `is2taddr` takes a host and service, and then consults the *isoentities*(5) file described in Chapter 7 of *Volume One* to construct a transport address.

```
struct TSAPaddr *is2taddr (host, service, is)
char    *host,
        *service;
struct isoservent *is;
```

4.1.1 Calling Address

Certain users of the transport service might need to know the name of the local host when they initiate a connection. The routine `TLocalHostName` has been provided for this reason.

```
char    *TLocalHostName ()
```

```
#include <isode/tsap.h>
#include <isode/isoservent.h>

...

register struct TSAPaddr *ta;
register struct isoservent *is;

...

if ((is = getisoserventbyname ("session", "tsap")) == NULL)
   error ("tsap/session");

/* RemoteHost is the host we're interested in,
   e.g., "gremlin.nrtc.northrop.com" */

if ((ta = is2taddr (RemoteHost, NULLCP, is)) == NULLTA)
   error ("address translation failed");

...
```

Figure 4.1: Constructing the TSAP address for the Session provider

4.1.2 Address Encodings

It may be useful to encode a transport address for viewing. Although a consensus for a standard way of doing this has not yet been reached, the routines `taddr2str` and `str2taddr` may be used in the interim.

```
char   *taddr2str (ta)
struct TSAPaddr *ta;
```

The parameter to this procedure is:

`ta`: the transport address.

If `taddr2str` fails, it returns the manifest constant `NULLCP`.

The routine `str2taddr` takes an ascii string encoding and returns a transport address.

```
struct TSAPaddr *str2taddr (str)
char   *str;
```

The parameter to this procedure is:

`str`: the ascii string.

If `str2taddr` fails, it returns the manifest constant `NULLTA`.

Once a connection has been established, the routine `TGetAddresses` may be called to retrieve to the associated addresses. (Normally this information is presented to the application during connection establishment.)

```
struct TSAPaddr *TGetAddresses (sd, initiating,
        responding, td)
int     sd;
struct TSAPaddr *initiating,
               *responding;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`initiating`: the TSAP address of the initiator (to be filled-in);

`responding`: the TSAP address of the responder (to be filled-in); and,

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

If the call is successful, the `initiating` and `responding` parameters will be updated accordingly. Otherwise, the parameter `td` contains the reason for the failure.

4.2 Connection Establishment

Until the connection has been fully established, the implementation distinguishes between clients and servers, which are more properly referred to as *initiators* and *responders*, to use the OSI terminology.

4.2.1 Connection Negotiation

From the user's perspective, there are two parameters which are negotiated by the transport providers during connection establishment: *expedited data transfer*, and the *maximum size* of transport service data units.

Expedited Data

If the transfer of expedited data is negotiated, then small amounts of data may be sent out-of-band once the connection has been established. The size of the largest discrete unit to be sent is `TX_SIZE` (which is non-negotiable). This parameter is negotiated downward; that is, both the initiator and responder must agree to the use of expedited data.

Maximum TSDU Size

The transport provider will accept arbitrarily large transport service data units (TSDUs) and transparently fragment and re-assemble them during transit. Hence, the actual TSDU is of unlimited size. However, for efficiency reasons, it may be desirable for the user to send TSDUs which are no larger than a certain threshold. When a connection has been established, the service providers inform the initiator and responder as to what this threshold is.

NOTE: In the current implementation, TSDUs which are no larger than the maximum atomic TSDU size are handled very efficiently. For optimal performance, users of the transport service should strive to avoid sending TSDUs which are larger than this threshold.

4.2.2 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 system implements the desired TSAP entity. For efficiency reasons, the *tsapd* program contains the bootstrap for several providers (e.g., session).

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 TSAP state contained in these magic arguments. This is done by calling the routine `TInit`, which on a successful return, is equivalent to a T-CONNECT.INDICATION event from the transport service provider.

```
int     TInit (vecp, vec, ts, td)
int     vecp;
char  **vec;
struct TSAPstart *ts;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

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

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

If `TInit` is successful, it returns information in the `ts` parameter, which is a pointer to a `TSAPstart` structure.

```
struct TSAPstart {
    int     ts_sd;

    struct TSAPaddr ts_calling;
    struct TSAPaddr ts_called;

    int     ts_expedited;

    int     ts_tsdusize;

    struct QOStype ts_qos;

#define TS_SIZE         32
    int     ts_cc;
    char    ts_data[TS_SIZE];
};
```

The elements of this structure are:

`ts_sd`: the transport-descriptor to be used to reference this connection;

`ts_calling`: the address of peer initiating the connection;

`ts_called`: the address of the peer being asked to respond;

`ts_expedited`: whether initiator requests the use of expedited data;

`ts_tsdusize`: the largest atomic TSDU size that can be used on the connection (see the note on page 106);

`ts_qos`: the quality of service on the connection (see Section 4.6.2); and,

`ts_data/ts_cc`: any initial data (and the length of that data).

If the call to `TInit` is not successful, then the user is informed that a T-DISCONNECT.INDICATION event has occurred, and the relevant information is returned in a `TSAPdisconnect` structure.

```
struct TSAPdisconnect {
    int     td_reason;

#define TD_SIZE         64
    int     td_cc;
    char    td_data[TD_SIZE];
};
```

The elements of this structure are:

`td_reason`: reason for disconnect (codes are listed in Table 4.1); and,

`td_data/td_cc`: any disconnect data (and the length of that data) from the peer.

After examining the information returned by `TInit` on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the connection. If accepting, the `TConnResponse` routine is called (which corresponds to the T-CONNECT.RESPONSE action).

```
int     TConnResponse (sd, responding, expedited, data, cc,
                       qos, td)
int     sd;
struct TSAPaddr *responding;
int     expedited,
        cc;
char   *data;
struct QOStype *qos;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`responding`: the TSAP address of the responder (defaulting to the called address, if not present);

`expedited`: whether expedited data is to be permitted (this value must be `0` unless the T-CONNECT.INDICATION event indicated a willingness on the part of the initiator to support expedited data transfer);

Provider-initiated Disconnect (fatal)	
`DR_NORMAL`	Normal disconnect by session entity
`DR_REMOTE`	Remote transport entity congested at connect request time
`DR_CONNECT`	Connection negotiation failed
`DR_DUPLICATE`	Duplicate source reference detected for the same pair of NSAPs
`DR_MISMATCH`	Mismatched references
`DR_PROTOCOL`	Protocol error
`DR_OVERFLOW`	Reference overflow
`DR_REFUSED`	Connect request refused on this network connection
`DR_LENGTH`	Header or parameter length invalid
`DR_NETWORK`	Network disconnect
User-initiated Disconnect (fatal)	
`DR_UNKNOWN`	Reason not specifed
`DR_CONGEST`	Congestion at TSAP
`DR_SESSION`	Session entity not attached to TSAP
`DR_ADDRESS`	Address unknown
Interface Errors (non-fatal)	
`DR_PARAMETER`	Invalid parameter
`DR_OPERATION`	Invalid operation
`DR_TIMER`	Timer expired
`DR_WAITING`	Indications waiting

Table 4.1: TSAP Failure Codes

data/cc: any initial data (and the length of that data, which may not exceed TC_SIZE octets);

qos: the quality of service on the connection (see Section 4.6.2); and,

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

If the call to TConnResponse is successful, then connection establishment has completed and the users of the transport service now operate as symmetric peers. Otherwise, if the call fails and the reason is not an interface error (see Table 4.1 on page 109), then the connection is closed.

If instead, the responder wishes to reject the connection, it should fire the T-DISCONNECT.REQUEST action by calling the TDiscRequest routine.

```
int     TDiscRequest (sd, data, cc, td)
int     sd;
char   *data;
int     cc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

sd: the transport-descriptor;

data/cc: any disconnect data (and the length of that data, which may not exceed TD_SIZE octets); and,

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

After a return from this call, the responder may exit.

4.2.3 Client Initialization

A program wishing to connect to another user of transport services calls the TConnRequest routine, which corresponds to the T-CONNECT.REQUEST action.

```
int     TConnRequest (calling, called, expedited, data, cc,
                qos, tc, td)
struct TSAPaddr *calling,
                *called;
int     expedited,
        cc;
char   *data;
struct QOStype *qos;
struct TSAPconnect *tc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`calling`: the TSAP address of the initiator; (need not be present);

`called`: the TSAP address of the responder;

`expedited`: whether expedited data is to be permitted;

`data/cc`: any initial data (and the length of that data, which may not exceed `TS_SIZE` octets);

`qos`: the quality of service on the connection (see Section 4.6.2);

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

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

If the call to `TConnRequest` is successful (a successful return corresponds to a T-CONNECT.CONFIRMATION event), then information is returned in the `tc` parameter, which is a pointer to a `TSAPconnect` structure.

```
struct TSAPconnect {
    int     tc_sd;

    struct TSAPaddr tc_responding;

    int     tc_expedited;
```

```
    int     tc_tsdusize;

    struct QOStype tc_qos;

#define TC_SIZE         32
    int     tc_cc;
    char    tc_data[TC_SIZE];
};
```

The elements of this structure are:

`tc_sd`: the transport-descriptor to be used to reference this connection;

`tc_responding`: the responding peer's address (which is the same as the `called` address given as a parameter to `TConnRequest`);

`tc_expedited`: whether expedited data will be supported;

`tc_tsdusize`: the largest atomic TSDU size that can be used on the connection (see the note on page 106);

`tc_qos`: the quality of service on the connection (see Section 4.6.2); and,

`tc_data/tc_cc`: any initial data (and the length of that data).

If the call to `TConnRequest` is successful, then connection establishment has completed and the users of the transport service now operate as symmetric peers. Otherwise, if the call fails then the connection is not established, and the `TSAPdisconnect` structure has been updated.

Asynchronous Connections

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

```
int     TAsynConnRequest (calling, called, expedited,
                data, cc, qos, tc, td, async)
struct TSAPaddr *calling,
```

```
                *called;
int     expedited,
        cc,
        async;
char   *data;
struct QOStype *qos;
struct TSAPconnect *tc;
struct TSAPdisconnect  *td;
```

The additional parameter to this procedure is:

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

If the `async` parameter is non-zero, then `TAsynConnRequest` returns one of four values: `NOTOK`, which indicates that the connection request failed; `DONE`, which indicates that the connection 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 `TConnRequest` are employed (i.e., a `NOTOK` return from `TAsynConnRequest` is equivalent to a `NOTOK` return from `TConnRequest`, and, a `DONE` return from `TAsynConnRequest` is equivalent to a `OK` return from `TConnRequest`). In the final case, when either `CONNECTING_1` or `CONNECTING_2` is returned, only the `tc_sd` element of the `tc` parameter has been updated; it reflects the transport-descriptor to be used to reference this connection. Note that the `data` parameter is still being referenced by *libtsap*(3n) and should not be tampered with until the connection attempt has been completed.

To determine when the connection attempt has been completed, the routine `xselect` (consult Section 2.4 of *Volume One*) should be used after calling `TSelectMask`. In order to determine if the connection attempt was successful, the routine `TAsynRetryRequest` is called:

```
int     TAsynRetryRequest (sd, tc, td)
int     sd;
struct TSAPconnect *tc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

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

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

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

In order to make efficient use of the asychronous connection facility, it is necessary to understand a bit of its underlying mechanisms. From a temporal perspective, connection establishment consists of two parts:

1. establishing a reliable end-to-end connection; and,

2. exchanging connection establishment information.

In some cases, the underlying transport mechanisms accomplish both simultaneously (when the end-to-end connection is built, connection establishment information is also exchanged).

Thus, in order to to perform asynchronous connections effectively, use of `TAsynConnRequest` and `TAsynRetry` should reflect this two-step process:

1. Call `TAsynConnRequest` with the `async` parameter taking the value 1. If the return value was either `NOTOK` (the connection was not established), or `DONE` (the connection was established), then terminate this algorithm.

   Otherwise, a return value of `CONNECTING_1` or `CONNECTING_2` indicates that connection establishment process has begun. Remember this value.

2. At some point in the future, call `TSelectMask` to get an argument for `xselect`. Then call `xselect` checking to see if writing is permitted. Then call `xselect` checking for writing if the remembered value was `CONNECTING_1` and for reading if it was `CONNECTING_2`. If either call returns `NOTOK`, then a catastrophic error has occurred.

   Repeat this step as often as necessary until `xselect` says that reading or writing as required is permitted.

3. Call `TAsynRetry`. If the return value was either `NOTOK` (the connection was not established), or `DONE` (the connection was established), then terminate this algorithm.

   Otherwise, a return value of `CONNECTING_1` or `CONNECTING_2` indicates that connection establishment is *still* in progress. Remember this value and go back to the previous step.

Although this seems complicated, implementation of these rules is actually straight-forward. In most cases, your code will do some work unrelated to the connection

Note that this procedure is equally applicable to the higher-layers (session, presentation, and association control) which also provide asynchronous connection facilities. For example, at the application layer, the routines `AcAsynAssocRequest` and `AcAsynRetryRequest` would be used.

4.3 Data Transfer

Once the connection has been established, a transport-descriptor is used to reference the connection. This is usually the first parameter given to any of the remaining routines in the *libtsap*(3n) library. Further, the last parameter is usually a pointer to a `TSAPdisconnect` structure (as described on page 108). If a call to one of these routines fails, then the structure is updated. Otherwise, if the value of the `td_reason` element is associated with a fatal error, then the connection is closed. That is, a T-DISCONNECT.INDICATION event has occurred. The `DR_FATAL` macro can be used to determine this.

```
int     DR_FATAL (r)
int     r;
```

For protocol purists, the `DR_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     DR_OFFICIAL (r)
int     r;
```

4.3.1 Sending Data

There are three routines that may be used to send data. A call to the `TDataRequest` routine is equivalent to a T-DATA.REQUEST action on the part of the user.

```
int     TDataRequest (sd, data, cc, td)
int     sd;
char   *data;
int     cc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`data/cc`: the data to be written (and the length of that data); and,

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

If the call to `TDataRequest` is successful, then the data has been queued for sending. Otherwise, the `td` parameter indicates the reason for failure.

The `TWriteRequest` routine is similar in nature to the `TDataRequest` routine, but uses a different set of parameters. The invocation is:

```
int     TWriteRequest (sd, uv, td)
int     sd;
struct udvec *uv;
struct TSAPdisconnect  *td;
```

While the parameters are:

`sd`: the transport-descriptor;

`uv`: the data to be written, described in a null-terminated array of scatter/gather elements:

```
struct udvec {
    caddr_t uv_base;
    int     uv_len;
};
```

The elements of the structure are:

uv_base: the base of an element; and,

uv_cc: the length of an element.

and,

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

If the call to TWriteRequest is successful, then the data has been queued for sending. Otherwise, the td parameter indicates the reason for failure.

A call to the TExpdRequest routine is equivalent to a T-EXPEDITED-DATA.REQUEST action on the part of the user.

```
int     TExpdRequest (sd, data, cc, td)
int     sd;
char   *data;
int     cc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

sd: the transport-descriptor;

data/cc: the data to be written (and the length of that data, which may not exceed TX_SIZE octets); and,

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

If the call to TExpdRequest is successful, then the data has been queued for expedited sending. Otherwise, the td parameter indicates the reason for failure.

4.3.2 Receiving Data

There is one routine that is used to read data, TReadRequest, a call to which is equivalent to waiting for a T-DATA.INDICATION or T-EXPEDITED-DATA.INDICATION event.

```
int     TReadRequest (sd, tx, secs, td)
int     sd;
struct TSAPdata *tx;
int     secs;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`tx`: a pointer to the `TSAPdata` structure to be given the data;

`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,

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

If the call to `TReadRequest` is successful, then the data has been read into the `tx` parameter.

```
struct TSAPdata {
    int     tx_expedited;

    int     tx_cc;
    struct qbuf tx_qbuf;
};
```

The elements of a `TSAPdata` structure are:

`tx_expedited`: whether the data was received via expedited transfer (i.e., an T-EXPEDITED-DATA.INDICATION event occurred);

`tx_cc`: the total number of octets that was read; and,

`tx_qbuf`: the data that was read, in a buffer-queue form.

```
        struct qbuf {
            struct qbuf *qb_forw;
            struct qbuf *qb_back;
```

```
        int    qb_len;
        char  *qb_data;

        char   qb_base[1];
    };
```

The elements of a `qbuf` structure are:

`qb_forw/qb_back`: forward and back pointers;

`qb_data/qb_len`: the user data (and the length of that data); and,

`qb_base`: the extensible array containing the data.

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

```
void    TXFREE (tx)
struct TSAPdata *tx;
```

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

NOTE:	Because the `TSAPdata` structure contains a `qbuf` element, care must be taken in initializing and copying variables of this type. The routines in *libtsap*(3n) library will correctly initialize these structures when given as parameters. But, users who otherwise manipulate `TSAPdata` structures should take great care.

Otherwise if the call to `TReadRequest` did not succeed, the `td` parameter indicates the reason for failure.

4.3.3 Asynchronous Event Handling

The data transfer events discussed thus far have been synchronous in nature. Some users of the transport service may wish an asynchronous interface.

The `TSetIndications` routine is used to change the service associated with a transport-descriptor to or from an asynchronous interface.

```
int     TSetIndications (sd, data, disc, td)
int     sd;
int   (*data) (),
      (*disc) ();
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`data`: the address of an event-handler routine to be invoked when data has arrived (either a T-DATA.INDICATION or T-EXPEDITED-DATA.INDICATION event occurs);

`disc`: the address of an event-handler routine to be invoked when the connection has been closed (a T-DISCONNECT.INDICATION event occurs); and,

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

If the service is to be made asynchronous, then both `data` and `disc` are specified; otherwise, if the service is to be made synchronous, neither should be specified (use the manifest constant `NULLIFP`). The most likely reason for the call failing is `DR_WAITING`, which indicates that an event is waiting for the user.

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.

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

```
(*data) (sd, tx);
int     sd;
struct TSAPdata *tx;
```

The parameters are:

`sd`: the transport-descriptor; and,

`tx`: a pointer to a `TSAPdata` structure containing the data.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released with the `TXFREE` macro (described on page 119) when no longer needed.

Similarly, when an event associated with connection release occurs, the event-handler is also invoked with two parameters:

```
(*disc) (sd, td);
int     sd;
struct TSAPdisconnect *td;
```

The parameters are

`sd`: the transport-descriptor; and,

`td`: a pointer to a `TSAPdisconnect` structure indicating why the connection was released.

Note that the transport-descriptor is no longer valid at the instant the call is made.

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

4.3.4 Synchronous Event Multiplexing

A user of the transport service may wish to manage multiple transport-descriptors simultaneously; the routine `TSelectMask` is provided for this purpose. This routine updates a file-descriptor mask and associated counter for use with `xselect`.

```
int     TSelectMask (sd, mask, nfds, td)
int     sd;
fd_set *mask,
int    *nfds;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

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

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

`td`: a pointer to a `TSAPdisconnect` 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 `DR_WAITING`, which indicates that an event is waiting for the user.

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

In addition, the routine `TSelectOctets` is provided to return an estimate of how many octets might be returned by the next call to `TReadRequest`:

```
int     TSelectOctets (sd, nbytes, td)
int     sd;
long   *nbytes;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`nbytes`: a pointer to a longword location that, on success, will be updated to contain the number of octets that might be returned; and,

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

4.4 Connection Release

The `TDiscRequest` routine is used to release a connection. Note, that the TSAP makes no guarantee that any queued data will be received before the connection closes.

```
int     TDiscRequest (sd, data, cc, td)
int     sd;
char   *data;
int     cc;
struct TSAPdisconnect  *td;
```

The parameters to the procedure are described on page 110.

4.5 State Saving and Restoration

Some users of the transport service, and in particular the session provider, require the ability to *execve*(2) another process image and have that process use the transport connection. Since the *libtsap*(3n) library is not kernel-resident, special provisions are necessary to support this behavior.

4.5.1 Saving the State

The routine `TSaveState` is used to record the state of the TSAP for a given transport-descriptor.

```
int     TSaveState (sd, vec, td)
int     sd;
char  **vec;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`vec`: a pointer to the first free slot in the argument vector for *execve*(2); and,

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

If the call succeeds, then an extra *magic argument* has been placed in the argument vector. The most likely reason for the call failing is `DR_WAITING`, which indicates that an event is waiting for the user.

NOTE:	Once a successful call to `TSaveState` is made on a transport descriptor, that descriptor may no longer be referenced until a corresponding call to `TRestoreState` is successful.

4.5.2 Restoring the State

The routine `TRestoreState` is used to re-initialize the state of the TSAP.

```
int     TRestoreState (buffer, ts, td)
char   *buffer;
struct TSAPstart  *ts;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`buffer`: the *magic argument* constructed by `TSaveState`;

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

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

If `TRestoreState` is successful, it returns information in a `TSAPstart` structure, as defined on page 107. There is one exception however: in the current implementation the `ts_cc` and `ts_data` elements are undefined on a successful return from `TRestoreState`.

4.6 Cookie Parameters

There are two *cookie* parameters: network addresses and quality of service.

4.6.1 Network Addresses

Network addresses can vary greatly. In this software distribution, a “unified” format has been adopted in the NSAPaddr structure:

```
struct NSAPaddr {
    long    na_stack;
#define NA_NSAP 0
#define NA_TCP  1
#define NA_X25  2
#define NA_BRG  3

    long    na_community;

    union {
        struct na_nsap {
#define NASIZE  64
            char    na_nsap_address[NASIZE];
            char    na_nsap_addrlen;
        }               un_na_nsap;

        struct na_tcp {
#define NSAP_DOMAINLEN  63
            char    na_tcp_domain[NSAP_DOMAINLEN + 1];
            u_short na_tcp_port;
            u_short na_tcp_tset;
#define NA_TSET_TCP    0x0001
#define NA_TSET_UDP    0x0002
        }               un_na_tcp;

        struct na_x25 {
#define NSAP_DTELEN     15
            char    na_x25_dte[NSAP_DTELEN + 1];
            char    na_x25_dtelen;

#define NPSIZE  4
            char    na_x25_pid[NPSIZE];
            char    na_x25_pidlen;

#define CUDFSIZE 16
```

```
                char    na_x25_cudf[CUDFSIZE];
                char    na_x25_cudflen;

#define FACSIZE 6
                char    na_x25_fac[FACSIZE];
                char    na_x25_faclen;
            }               un_na_x25;
        }           na_un;

#define na_address      na_un.un_na_nsap.na_nsap_address
#define na_addrlen      na_un.un_na_nsap.na_nsap_addrlen

#define na_domain       na_un.un_na_tcp.na_tcp_domain
#define na_port         na_un.un_na_tcp.na_tcp_port

#define na_dte          na_un.un_na_x25.na_x25_dte
#define na_dtelen       na_un.un_na_x25.na_x25_dtelen
#define na_pid          na_un.un_na_x25.na_x25_pid
#define na_pidlen       na_un.un_na_x25.na_x25_pidlen
#define na_cudf         na_un.un_na_x25.na_x25_cudf
#define na_cudflen      na_un.un_na_x25.na_x25_cudflen
#define na_fac          na_un.un_na_x25.na_x25_fac
#define na_faclen       na_un.un_na_x25.na_x25_faclen
};
#define NULLNA  ((struct NSAPaddr *) 0)
```

As shown, this structure is really a discriminated union (a structure with a tag element followed by a union). Based on the value of the tag (`na_stack`), a different structure is selected.

For a native OSI CO-mode transport service, the value of the tag is `NA_NSAP`, and the following elements are meaningful:

`na_address/na_addrlen`: the network address (and its length), binary-valued.

For emulation of the OSI transport service on top of the TCP, the value of the tag is `NA_TCP`, and the following elements are meaningful:

`na_domain`: the null-terminated domain name (e.g., "gonzo.twg.com") or dotted-quad (e.g., "128.99.0.17");

`na_port`: the TCP-port number offering the service (if zero, the service on port 102 is used); and,

`na_tset`: the set of IP-based transport services available at the address (if zero, the TCP service is used).

For use of a single-subnet X.25, the value of the tag is `NA_X25`, and the following elements are meaningful:

`na_dte/na_dtelen`: the X.121 address (and its length), ascii-valued, possibly null-terminated;

`na_pid/na_pidlen`: the protocol id (and its length), binary-valued;

`na_cudf/na_cudflen`: the call user data (and its length), binary-valued; and,

`na_fac/na_faclen`: the negotiated facilities proposed in the call request packet (and its length), binary-valued.

For use of a TP0-bridge between the TCP and X.25, the value of the tag is `NA_BRG`, and the elements above are meaningful.

If the value of the tag is not `NA_NSAP`, it may be useful to normalize the address into a "real" OSI address. The routine `na2norm` is used:

```
char   *na2norm (na)
struct NSAPaddr *na;
```

The parameter to this procedure is:

`na` : the network address to be normalized.

A new network address is returned from a static area which contains the normalized form.

The routine `na2str` takes a network address and returns a null-terminated ascii string suitable for viewing:

```
char   *na2str (na)
struct NSAPaddr *na;
```

The parameter to this procedure is:

`na` : the network address to be printed.

The `na_community` field is an internal number used to distinguish between different OSI communities. Consult Chapter 8 starting on page 167 for further details.

4.6.2 Quality of Service

Currently, quality of service is largely uninterpreted by the software. However, the quality of service structure contains those parameters which are supported:

```
struct QOStype {
                                /* transport QOS */
    int     qos_reliability;
#define HIGH_QUALITY    0
#define LOW_QUALITY     1

                                /* session QOS */
    int     qos_sversion;
    int     qos_extended;
    int     qos_maxtime;
};
#define NULLQOS ((struct QOStype *) 0)
```

The elements of this structure are:

`qos_reliability`: the "reliability" level of the connection, either high- or low-quality;

`qos_sversion`: the session version requested/negotiated on this connection (only applicable above the transport layer, obviously), if the manifest constant `NOTOK` is used when initiating a connection, this indicates that the highest possible version should be negotiated;

`qos_extended`: the extended control parameter for the connection, (if non-zero, extended control is used by the session layer); and,

`qos_maxtime`: after a transport connection is established, the maximum number of seconds to wait for an acknowledgement from the responding SPM (any non-positive number indicates that no time-limit is desired).

4.7 Listen Facility

The *libtsap*(3n) library, supports a facility which permits a process to *listen* for certain connections. This can be useful for implementing an application which requires that a single server process handle multiple clients, or for connection recovery. These routines return the manifest constant `NOTOK` on error, and `OK` on success, they also update the `td` parameter given to the routine. This parameter is a pointer to a `TSAPdisconnect` structure.

Although this facility is described in terms of the *libtsap*(3n) library, it will function at any other higher-layer in the system (e.g., the listen facility can be used for session or application-entities).

A program starts listening for an particular connection by calling the routine `TNetListen`.

```
int     TNetListen (ta, td)
struct TSAPaddr *ta;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`ta`: the transport address (0 or more network addresses) to listen on; and,

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

If the call is successful, then the program is now listening for incoming connections on that network address. Otherwise, the `td` parameter indicates the reason for failure.

A variant of `TNetListen` is the `TNetUnique` routine, which starts the process listening on a set of unique network (sub)addresses.

```
int     TNetUnique (ta, td)
struct TSAPaddr *ta;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`ta`: a transport address containing one or more partially filled-in network addresses; and,

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

If the call is successful, each network address in the `ta` parameter is fully filled-in, the program is now listening for incoming connections on the resulting transport. Otherwise, the `td` parameter indicates the reason for failure.

To check when a new connection is waiting, or when existing connections have activity on them, the routine `TNetAccept` is used.

```
int     TNetAccept (vecp, vec, nfds, rfds, wfds, efds, secs,
                td)
int    *vecp,
char  **vec;
int     nfds;
fd_set *rfds,
       *wfds,
       *efds;
int     secs;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`vecp/vec`: the initialization vector for the new connection.

`nfds/rfds/wfds/efds`: connection-descriptors for use with `xselect`;

`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 a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call to `TNetAccept` succeeds then the value of `vecp` should be checked. If `vecp` is greater than zero, a new connection has been accepted, and `TInit` should be called, presumably followed by `TConnResponse`.[1] Regardless of the value of `vecp`, the value of `rfds` and `wfds` should be checked to see which

[1]Actually, any service addressable via a transport selector can use this service, e.g., if appropriate, a call to `AcInit`, followed by a call to `AcAssocResponse`, can be made.

connections have activity pending. For these connections, any reads should probably be done with a `secs` argument indicating a polling operation (i.e., a value of `OK`). Otherwise, if the call to `TNetAccept` fails, then the `td` parameter indicates the reason for failure.

NOTE:	The `TNetAccept` procedure when first called arranges to clean up dead child processes. If the program will run any subprocesses and check their exit status, the automatic collection of zombie process should be disabled, by first calling `TNetAccept` with a timeout of `OK` and then setting the child signal handler to it's default state.

The routine `TNetAccept` is actually a macro which invokes a routine called `TNetAcceptAux`:

```
int     TNetAcceptAux (vecp, vec, newfd, ta, nfds, rfds, wfds,
                efds, secs, td)
int    *vecp,
char  **vec;
int     newfd;
struct TSAPaddr *ta;
int     nfds;
fd_set *rfds,
       *wfds,
       *efds;
int     secs;
struct TSAPdisconnect *td;
```

The additional parameters to this procedure are:

`newfd`: a pointer to an integer which will be given the value of the connection-descriptor associated with the new connection; and,

`ta`: a pointer to a `TSAPaddr` structure which will be given the value of the transport address receiving the new connection (the called or listening address).

Prior to exiting, the user should call `TNetClose` to stop listening for connections.

```
int     TNetClose (ta, td)
struct TSAPaddr *ta;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`ta`: the transport address to stop listening on (use the manifest constant `NULLTA` to stop listening on all addresses); and,

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

If the call is successful, then the program has stopped listening for incoming connections on that network address. Otherwise, the `td` parameter indicates the reason for failure.

4.8 Queued (non-blocking) Writes Facility

All "read" operations in the ISODE are inherently non-blocking. Historically, "write" operations have not required this capability. However, some applications (e.g., the QUIPU DSA) require non-blocking writes. The routine `TSetQueuesOK` is used to enable or disable this facility:

```
int     TSetQueuesOK (sd, onoff, td)
int     sd;
int     onoff;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`onoff`: a flag indicating whether the facility is to be enabled (non-zero value) or disabled (zero value) for the transport-descriptor; and,

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

If the call is not successful, the `td` parameter indicates the reason for failure. Otherwise, if the `onoff` parameter has a non-zero value, then any "write" operations which ultimately map to this transport-descriptor will not block the process. This is done by maintaining a queue of write operations and periodically retrying them. In order to schedule retries, the routine `TNetAccept` described earlier should be called frequently. In addition, if some failure occurs during the retry (e.g., the transport connection is disconnected), then `TNetAccept` will mark that descriptor as being ready for reading. When the program interrogates the descriptor, the appropriate error code will be returned. Note that in order for this action to occur, any descriptor which has queued writes enabled must be included in the `rfs` parameter supplied when the `TNetAccept` routine is called.

4.9 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `td` parameter given to the routine. The `td_reason` element of the `TSAPdisconnect` structure can be given as an parameter to the routine `TErrString` which returns a null-terminated diagnostic string.

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

4.10 Compiling and Loading

Programs using the *libtsap*(3n) library should include `<isode/tsap.h>`. These programs should also be loaded with `-ltsap` and, for reasons explained momentarily, `-licompat`.

4.11 An Example

Let's consider how one might construct a loopback entity that resides on the TSAP. This entity will use a synchronous interface.

First, we must decide at what address the entity will reside. For simplicity's sake, we'll say that the location is `tsap/echo`, as defined in the *isoservices*(5) database.

Next, we actually code the loopback entity. There are two parts to the program: initialization and data transfer; release will occur whenever data transfer fails. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure 4.2, the initialization steps for the loopback entity, including the outer C wrapper, is shown. The entity does not examine any of its arguments, but could do so after the call to `TInit`. After examining the arguments, it could decide to reject the connection attempt, by using `TDiscRequest`. Instead, it uses `TConnResponse` with the exact negotiated parameters with which it was supplied. Hence, if the initiator wanted to use expedited data transfer, the loopback entity responding to the connection would honor that.

In Figure 4.3 on page 136, the data transfer loop is realized. The loopback entity awaits an event from the service provider, which either indicates that data has arrived or that the connection has been closed. If a disconnection occurred (a T-DISCONNECT.INDICATION event is reported), then the reason is checked. If the event did not occur because the initiator performed a T-DISCONNECT.REQUEST indication, then an error is signaled. Otherwise, the inner-loop is terminated and the process will gracefully terminate. If instead data arrived, it is echoed back to the initiator.

```
#include <stdio.h>
#include <isode/tsap.h>
#include <isode/isoservent.h>


/* ARGSUSED */

main (argc, argv, envp)
int     argc;
char  **argv,
      **envp;
{
    int     result,
            sd;
    struct TSAPstart    tss;
    register struct TSAPstart  *ts = &tss;
    struct TSAPdata txs;
    register struct TSAPdata  *tx = &txs;
    struct TSAPdisconnect   tds;
    register struct TSAPdisconnect *td = &tds;

    if (TInit (argc, argv, ts, td) == NOTOK)
        error ("T-CONNECT.INDICATION: %s", TErrString (td -> td_reason));
    sd = ts -> ts_sd;

/* examine argv here, if need be */

    if (TConnResponse (sd, &ts -> ts_called, ts -> ts_expedited,
                ts -> ts_data, ts -> ts_len, NULLQOS, td) == NOTOK)
        error ("T-CONNECT.RESPONSE: %s", TErrString (td -> td_reason));

...
```

Figure 4.2: Initializing the loopback entity

```
...

    for (;;) {
        if (TReadRequest (sd, tx, NOTOK, td) == NOTOK) {
            if (td -> td_reason != DR_NORMAL)
                error ("T-READ.REQUEST: %s", TErrString (td -> td_reason));
            break;
        }

        if (tx -> tx_expedited)
            result = TExpdRequest (sd, tx -> tx_base, tx -> tx_cc, td);
        else
            result = TDataRequest (sd, tx -> tx_base, tx -> tx_cc, td);
        if (tx -> tx_base)
            free (tx -> tx_base);

        if (result == NOTOK)
            error ("%s:  %s", tx -> tx_expedited ? "T-EXPEDITED-DATA.REQUEST"
                        : "T-DATA.REQUEST", TErrString (td -> td_reason));

    }

    exit (0);
}
```

Figure 4.3: Data Transfer for the loopback entity

4.12 Compatibility Issues

The *libicompat*(3) library is used as an aid for porting the software from one system to another. This library contains generic service routines, which are in turn composed of the native facilities available on the target host. All of the higher layer `#include` files automatically reference parts of this library as appropriate. Hence, when loading the the portions of the software independently, the loader must be given the `-licompat` flag.

The problem of this approach, of course, is that not all facilities can be precisely emulated. To misquote M.A. Padlipsky[Pad85]:

> *Sometimes when you try to make an apple look like an orange you get back something that smells like a lemon.*

4.13 For Further Reading

The ISO specification for transport services is defined in [ISO86]. The corresponding CCITT recommendation is defined in [CCI84c]. The document describing how these services can be implemented on top of the TCP[Pos81] is [RC87].

4.14 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.

The `na_type` and `na_subnet` fields of the `NSAPaddr` structure are now called `na_stack` and `na_community` respectively. For compatibility, macros are provided. These macros will be removed after this release.

Part III

Databases

Chapter 5

The ISODE Services Database

The database *isoservices* in the ISODE `ETCDIR` directory (usually */usr/etc/*) contains a simple mapping between textual descriptions of services, service selectors, and local programs.

NOTE:	Use of this database is deprecated. Consult Chapter 10 on page 160 of *Volume One* for further information.

The database itself is an ordinary ASCII text file containing information regarding the known services on the host. Each line contains

- the name of an entity and the provider on which the entity resides;
- the selector used to identify the entity to the provider, interpreted as a:

  number, if the selector starts with a hash-mark (‘`#`’). More precisely, this denotes the so-called GOSIP method for denoting selectors, which uses a two octet, network byte-order representation.

  ascii string, if the selector appears in double-quotes (‘`"`’). The usual escape mechanisms can be used to introduce non-printable characters.

  octet string, if all else fails. The standard “explosion” encoding is used, each octet in the string is represented by a two-digit hexadecimal quantity.

and,

- the program and argument vector to *execve*(2) when the service is requested.

Blanks and/or tab characters are used to separate items. All items after the first two are interpreted as an argument vector. 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.

5.1 Accessing the Database

The *libicompat*(3n) library contains the routines used to access the database. These routines ultimately manipulate an `isoservent` structure, which is the internal form.

```
struct isoservent {
    char   *is_entity;
    char   *is_provider;

#define ISSIZE  64
    int     is_selectlen;
    char    is_selector[ISSIZE];

    char  **is_vec;
    char  **is_tail;
};
```

The elements of this structure are:

`is_entity`: the name of the entity;

`is_provider`: the name of the provider on which the entity resides;

`is_selectoris_selectlen`: the selector used to identify the entity to the provider (the element `is_port` is an alias for this concept, used to denote the entity to the provider by means of a two-octet number specified in network-byte order);

`is_vec`: the *execve*(2) vector; and,

`is_tail`: the next free slot in `is_vec`.

The routine `getisoservent` reads the next entry in the database, opening the database if necessary.

```
struct isoservent *getisoservent ()
```

It returns the manifest constant `NULL` on error or end-of-file.

The routine `setisoservent` opens and rewinds the database.

```
int     setisoservent (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 `endisoservent` closes the database.

```
int     endisoservent ()
```

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 `getisoserventbyname` maps textual descriptions into the internal form.

```
struct isoservent *getisoserventbyname (entity, provider)
char   *entity,
       *provider;
```

The parameters to this procedure are:

`entity`: the entity providing the desired service; and,

`provider`: the provider supporting the named `entity`.

On a successful return, the `isoservent` structure describing that service is returned. On failure, the manifest constant `NULL` is returned instead.

The routine `getisoserventbyselector` performs the inverse function.

```
struct isoservent *getisoserventbyselector (provider,
                        selector, selectlen)
char   *provider,
       *selector;
int     selectlen;
```

The parameters to this procedure are:

`provider`: the provider supporting the desired entity; and,

`selector/selectlen`: the selector on the provider where the desired entity resides.

On a successful return, an `isoservent` structure describing the entity residing on the provider is returned.

The routine `getisoserventbyport` performs a similar function.

```
struct isoservent *getisoserventbyport (provider, port)
char   *provider;
unsigned short port;
```

The parameters to this procedure are:

`provider`: the provider supporting the desired entity; and,

`port`: the port on the provider (in network-byte order) where the desired entity resides.

On a successful return, an `isoservent` structure describing the entity residing on the provider is returned.

Chapter 6

The ISODE Tailoring File

The file *isotailor* in the ISODE `ETCDIR` directory (usually */usr/etc/*) contains a simple run-time configuration mechanism for programs loaded with the `-lisode` library.

The file itself is an ordinary ASCII text file containing information regarding the known tailoring options. Each line contains the option's name, a colon, the option's value, and a newline. The sharp character ('`#`') at the beginning of a line indicates a commentary line.

6.1 Tailor Variables

The options available, along with default values and a description of their meanings, are now described.

6.1.1 Local Environment Tailoring

`localname`: The name of the localhost. If not set, depending on the TCP/IP implementation you are running, the system will be queried for this value.

`binpath`: Where user programs are found.

`sbinpath`: Where system programs are bound.

`etcpath`: Where configuration files are found.

(continued on the next page)

6.1.2 Logging Tailoring

`logpath`: Where log files are found.

xyz`level`: The debugging level for the *xyz* module, defaulting to `none`.

Option	Module
`compatlevel`	*libicompat*(3n)
`addrlevel`	various
`tsaplevel`	*libtsap*(3n)
`ssaplevel`	*libssap*(3n)
`psaplevel`	*libpsap*(3)
`psap2level`	*libpsap2*(3n)
`acsaplevel`	*libacsap*(3n)
`rtsaplevel`	*librtsap*(3n)
`rosaplevel`	*librosap*(3n)

Options are seperated by whitespace. The debugging options available are:

`fatal`: fatal errors;

`exceptions`: exceptional events;

`notice`: informational notices;

`pdus`: *xyz*PDU printing;

`trace`: program tracing;

`debug`: full debugging (not fully defined at this point); and,

`all`: all of the above.

Logging of levels other than `fatal`, `exceptions`, or `notice` are subject to conditional compilation (the `-DDEBUG` option must be used during compilation).

xyz`file`: The file to be used for *xyz*PDU tracing. The file is written in append mode. If the filename supplied is '`-`' (a single dash), then the diagnostic output is used instead.

Option	Default	Module
`compatfile`	`%d.log`	*libicompat*(3n)
`addrfile`	`%d.log`	various
`tsapfile`	`%d.tpkt`	*libtsap*(3n)
`ssapfile`	`%d.spkt`	*libssap*(3n)
`psapfile`	`%d.pe`	*libpsap*(3)
`psap2file`	`%d.ppkt`	*libpsap2*(3n)
`acsapfile`	`%d.acpkt`	*libacsap*(3n)
`rtsapfile`	`%d.rtpkt`	*librtsap*(3n)
`rosapfile`	`%d.ropkt`	*librosap*(3n)

The *getpid*(2) call is used to supply the value for `%d`.

6.1.3 Directory Services Tailoring

`ns_enable`: enables use of a "user-friendly nameservice" to perform name/address resolution. This takes the value either `"on"` or `"off"`. If `"on"`, then an OSI Directory-based service will be use. If the nameservice lookup fails, the stub-directory will be used as a fallback.

`ns_address`: the transport address of the nameservice. It is specified using the ISODE "string" format, e.g.,

```
Internet=wp.psi.net+17006
```

which indicates that the nameservice lives in the TCP/IP communications domain on TCP port `17006` at host `wp.psi.net`. The nameservice is accessed via the OSI CO-mode transport service, so other kinds of addresses (e.g., X.25 addresses can be used as well).

6.1.4 Transport Switch Tailoring

The use of these variables is more usefully described in Chapter 8.

`ts_stacks`: Specifies which configured TS-stacks are enabled. This is useful when multiple machines (with different interfaces) share the same executables. Options are seperated by whitespace:

`tcp`: RFC1006 over TCP/IP;

`x25`: TP0 over X.25;

`cons`: TP0 over CONS;

`bridge`: TP0 over the TP0–bridge;

`tp4`: TP4 over CLNP; and,

`all`: all of the above.

Using this method, the *isotailor* file is a normally symbolic link to */private/etc/isotailor*.

`ts_interim`: Defines new OSI communities. Each community is defined by a macro in the *i*somacros(5) file.

`ts_communities`: Specifies which OSI communities are attached (either directly or through a transport-service bridge). Options are seperated by whitespace:

`int-x25`: International X.25;

`janet`: UK JANET;

`internet`: the capital-I Internet;

`realns`: OSI Internet (ha, ha);

`localTCP`: the TCP loopback address; and,

`all`: all of the above OSI communities, along with those communities defined by the `ts_interim` variable;

For example, a site with an X.25 connection might be attached to the International X.25 network, but not the JANET. Thus `ts_stacks` would include "x25", and `ts_communities` would include "int-x25" but not "janet".

Note that the ordering of communities is important: network addresses will be tried in the order that their respective communities are listed with this variable.

`default_nsap_community`: the default community to be used for NSAP addresses.

`default_x25_community:` the default community to be used for X.25 (DTE) addresses.

`default_tcp_community:` the default community to be used for TCP (RFC1006) addresses.

Transport-Service Bridge

There are two variables that can be specified. One is used on hosts making use of the TS-bridge, the other is used by hosts which run the TS-bridge:

`tsb_communities` A list of pairs of values. The first of each value should be a community name as defined in the `ts_communities` variable. The second value of the pair should be a presentation address using the ISODE "string" format. When a call is to be placed and the network corresponds to one of the communities given here, then a call through the TS-bridge given in the second variable will be made automatically.

`tsb_default_address` This variable contains a string encoded presentation address which the TS-bridge will listen on by default. This should normally consist of a set of network addresses with no selectors present.

Consider the case of a host with access to both the Internet and the International X.25 network. This host might have this entry in its *isotailor* file:

```
tsb˙default˙address: Internet=sheriff+17004“—Int-X25(80)=23426020017299+PID+03018000
```

This tells the TS-bridge to listen on two network endpoints. Hosts in the Internet community wishing to reach the International X.25 community would have this entry in their *isotailor* file:

```
tsb_communities: int-x25 Internet=sheriff+17004
```

Similarly, hosts in the International X.25 community wishing to reach the Internet community, would have the entry:

```
tsb_communities: internet Int-X25(80)=23426020017299+PID+03018000
```

6.1.5 Interface Specific Tailoring

Some network implementations used by ISODE require ISODE to be tailored for their correct use.

General X.25 Tailoring

The following tailoring variables are generally applicable to X.25 networks.

`x25_local_dte` It is normally necessary for ISODE to know it's local DTE address. This variable is used to set this. The default is empty, i.e., do not set a calling address in call requests.

`x25_local_pid` It is normally necessary for ISODE to know the X.25 protocol ID to listen on This is specified in hex-notation, e.g., `03010100`.

`x25_intl_zero` Some Public Data Networks require that X.121 addresses be modified before being conveyed. If this variable has the value `on` then any addresses with a non-local DNIC will have a leading zero appended.

`x25_strip_dnic` If this variable has the value `on` then any address with a local DNIC will have this removed.

`x25_dnic_prefix` If you use either or both of the preceding two mechanisms then you must use this variable to inform ISODE of the local DNIC for your host.

`x25level`: Defines the level of logging to be used for X.25 statistics logging. (At present, only `notice` messages are generated.)

`x25file`: Defines the filename to be used for X.25 statistics logging.

SunLink X.25

The following variables are currently only supported by the SunLink X.25 interface. They control the X.25 Facilities that are requested or accepted.

`reverse_charge` If `0` then don't request/allow (initiator/responder) reverse charging. If `1` then request/allow reverse charging.

recvpktsize/sendpktsize Size of level 3 packets. Valid values are `0` (default size), `16`, `32`, `64`, `128`, `256`, `512`, `1024` (octets in decimal).

recvwndsize/sendwndsize Size of level 3 window. Valid values are `0` (default window size), `1`–`7` or `1`–`127` (decimal).

recvthruput/sendthruput Send/receive throughput.

0	default throughput		
3	75	8	2400
4	150	9	4800
5	300	10	9600
6	600	11	19200
7	1200	12	48000

Values in bps decimal.

cug_req If `0` then don't use closed user group, if `1` then use closed user group specified by `cug_index`.

cug_index Closed user group in decimal (`00`–`99`).

fast_select_type `0` = don't use/allow fast select. `1` = calling side — only accept clear in response to fast select, called side — send clear in response to fast select. `2` = clear or accept is valid response to fast select.

rpoa_req If `0` then don't request RPOA (Recognised Private Operating Agency) transit. If `1` then request RPOA transit.

rpoa If `rpoa_req` is `1` then this is RPOA transit group in decimal (`0000`–`9999`).

See the SunLink X.25 Programmer's Manual for further explanations of these facilities.

Camtec CCL

There is one tailoring variable for the Camtec X.25 when used with the socket abstraction.[1]

`x25_outgoing_port` selects the physical port on the Camtec card for outgoing calls. It may take the value `A`, `B` or `#`. `A` and `B` are the X.121 WAN ports and `#` is the IEEE 802.3 (Ethernet) port. Incoming calls will be accepted on any port.

Bridge X.25

There are several tailorable variables that can specified for the bridge connection. These are:

`x25_bridge_host` selects the host that runs the tp0bridge being used. This should be a TCP accessible host.

`x25_bridge_port` selects the TCP port that the tp0bridge will be listening on. The default for this is port 146 (an internet assigned number), which should be defined in `/etc/services`.

`x25_bridge_addr` the X.121 address of the remote host.

`x25_bridge_listen` the X.121 address that this host will be listening on for incoming calls via the bridge.

`x25_bridge_pid` the protocol id used for listening along with the above address. This is a set of eight hex digits.

`x25_bridge_discrim` selects the network layer to use. When attempting to place a call with the bridge code configured as well as real X.25, the string selects the interface to use. If the string is empty, the bridge will always be used. If it is set to "–" the bridge will not be used. If the string is anything else, it is compared against the initial portion of the called X.121 address. If there is a match then the bridge is used, otherwise the real interface is called.

[1] The old device level interface is no longer supported.

6.2 Accessing the Tailoring File

The tailoring file is read usually when a program attempts or accepts its first connection. The `-lisode` library does this by calling the routine `isodetailor`:

```
void    isodetailor (myname, wantuser)
char   *myname;
int     wantuser;
```

The parameters to this procedure are:

`myname`: the name that the program was invoked with (used by the logging package described in Chapter 7); and,

`wantuser`: if non-zero, then a user-specific tailoring file, with the name `~/.myname_tailor`, should be consulted.

Note that in order to ensure consistent logging it is **critical** that the call to `isodetailor` be the first call made to *any* of the ISODE routines.

To override the default location of the tailoring file, use the routine `isodesetailor`:

```
char   *isodesetailor (file)
char   *file;
```

The parameter to this procedure is:

`file`: the filename to be used instead of the default. Future versions of this routine might act differently.

The filename is interpreted relative to the `-lisode` system area. To override this, specify a anchored pathname (e.g., on UNIX, one which starts with `/` or `./`). The routine returns the name of the default tailoring file.

To set a tailoring variable from some other configuration file, the routines `isodesetvar` and `isodexport` are used:

```
int     isodesetvar (name, value, dynamic)
char   *name,
       *value;
int     dynamic;
```

When this routine is invoked, it acts as though

```
name: value
```

was found in the tailoring file. The `dynamic` parameter, if non-zero, indicates that `value` may be freed if a subsequent call to `isodesetvar` is made which overrides the previous value.

The `isodexport` routine is called after one or more calls to `isodesetvar`, it performs any post-processing necessary to resynchronize the tailoring facilities.

```
void    isodexport ()
```

Thus, to read a private tailoring file, `isodesetvar` should be called for each tailoring line. Then, `isodexport` should be called once to resynchronize things.

Finally, it may be necessary to access files in the `-lisode` system area. The routine `isodefile` takes an filename and returns an anchored pathname.

```
char   *isodefile (file, ispgm)
char   *file;
int ispgm;
```

The parameters to this procedure are:

`file`: the filename to be expanded; and,

`ispgm`: non-zero if the target file is an executable (otherwise it is a database of some kind).

This routine is actually a macro which invokes the routine `_isodefile`:

```
char   *_isodefile (path, file)
char   *path,
       *file;
```

The parameters to this procedure are:

`path`: the directory where the filename should be expanded; and,

`file`: the filename to be expanded.

Chapter 7

The ISODE Logging Facility

Although not a database mechanisms, per se, the ISODE logging facility is used to manipulate general logs: used by both the ISODE and programs which use the ISODE.

7.0.1 Data-Structures

There is one primary data-structure, the LLog:

```
typedef struct  ll_struct {
    char   *ll_file;

    char   *ll_hdr;
    char   *ll_dhdr;

    int     ll_events;
#define LLOG_NONE 0
#define LLOG_FATAL 0x01
#define LLOG_EXCEPTIONS 0x02
#define LLOG_NOTICE 0x04
#define LLOG_PDUS 0x08
#define LLOG_TRACE 0x10
#define LLOG_DEBUG 0x20
#define LLOG_ALL 0xff
#define LLOG_MASK \
    "\020\01FATAL\02EXCEPTIONS\03NOTICE\04PDUS\05TRACE\06DEBUG"
```

```
    int     ll_syslog;

    int     ll_msize;

    int     ll_stat;
#define LLOGNIL 0x00
#define LLOGCLS 0x01
#define LLOGCRT 0x02
#define LLOGZER 0x04
#define LLOGERR 0x08
#define LLOGTTY 0x10
#define LLOGHDR 0x20
#define LLOGDHR 0x40

    int     ll_fd;
} LLog;
```

The elements of this structure are:

ll_file: the name of the file to use for the log, unless an absolute pathname (e.g., */usr/tmp/logfile*) or an anchored pathname (e.g., *./logfile*), the name is interpreted relative to the the `logpath` directory in the ISODE tailoring file (see Chapter 6);

ll_hdr: the logging header which is usually set by one of the utility routines described below;

ll_hdr/ll_dhdr: the so-called dynamic header;

ll_events/ll_syslog: a bitmask describing the logging events which are interesting to this log, any combination of:

Value	Meaning
LLOG_FATAL	fatal errors
LLOG_EXCEPTIONS	exceptional events
LLOG_NOTICE	informational notices
LLOG_PDUS	PDU printing
LLOG_TRACE	program tracing
LLOG_DEBUG	full debugging

In addition, the values `LLOG_NONE` by itself refers to no events and `LLOG_ALL` refers to all events being of interest. For those systems with a *syslog*(3) routine, the `ll_syslog` element indicates if the event should be given to *syslog*(8) as well;

`ll_msize`: the maximum size of the log, in units of Kbytes (a non-positive number indicates no limit);

`ll_stat`: assorted switches, any combination of:

Value	Meaning
`LOGCLS`	keep log closed, except when writing
`LOGCRT`	create log if necessary
`LOGZER`	truncate log when limits reached
`LOGERR`	log closed due to (soft) error
`LOGTTY`	also log to stderr
`LOGHDR`	static header allocated
`LOGDHR`	dynamic header allcoated

`ll_fd`: the file-descriptor corresponding to the log.

7.1 Accessing the Log

Typically, logs are not opened or closed directly — when an entry is made to a log, the log is opened (if necessary), the entry is written, and (usually) the log is then closed.

To open a log associated with a `LLog` structure, the routine `ll_open` is used:

```
int     ll_open (lp)
LLog   *lp;
```

The parameter to this routine is:

`lp`: a pointer to a `LLog` structure.

The `ll_open` routine will open the log, creating the corresponding file (if necessary). Logs are created mode `0666`. If the name of the file to use for the log is "-", then the LLOGTTY option is enabled and no file is actually opened. When determining the actual name of the file to use, a "%d" in the name will be replaced by the process-id of the program opening the log. On failure, the manifest constant NOTOK is returned. Otherwise, the log is opened (and left open, regardless of the presence of the LLOGCLS option), and the manifest constant OK is returned.

To close a log, the routine ll_close is used:

```
int     ll_close (lp)
LLog   *lp;
```

The parameter to this routine is:

lp: a pointer to a LLog structure.

This routine returns the manifest constant OK on success (even if the log was already closed), or NOTOK otherwise.

7.1.1 Timestamps

One of the characteristics of a log is that it contains an informational timestamp for each entry. This timestamp contains the date and time of the log and also two "header" strings, a static header and a dynamic header. Normally, these strings are constructed from the name of the program or subsystem using the log. The routine ll_hdinit is used to initialize the static header:

```
void   ll_hdinit (lp, prefix)
LLog   *lp;
char   *prefix;
```

The parameters to this routine are:

lp: a pointer to a LLog structure; and,

prefix: the name of the program or subsystem using the log.

This routine will form a header consisting of the program name, the process-id, and the user-name.

The routine ll_dbinit is similar, but also enables debugging features:

```
void   ll_dbinit (lp, prefix)
LLog   *lp;
char   *prefix;
```

The parameters to this routine are:

`lp`: a pointer to a `LLog` structure; and,

`prefix`: the name of the program or subsystem using the log.

This routine will form a header identical to the one formed by `ll_hdinit`. It will then set the name of the file associated with the log to be relative to the current working directory. Finally, it turns on all event logging and logging to the user's terminal.

7.1.2 Making Log Entries

At the lowest level, the `ll_log` routine is used to append an entry to a log:

```
int     ll_log (lp, event, what, fmt, args ...)
LLog   *lp;
int     event;
char   *what,
       *fmt;
```

The parameters to this routine are:

`lp`: a pointer to a `LLog` structure;

`event`: the event type being logged (e.g., `LLOG_NOTICE`);

`what`: some text associated with a system call error (use the manifest constant `NULLCP` if the entry is not associated with an error in a system call); and,

`fmt/args`: an argument list to *printf*(3s).

The entry is only made if the log is enabled (in the `ll_events` field of the `LLog` structure) for the event listed as a parameter to `ll_log`. If there was a problem in writing to the log, `ll_log` returns the manifest constant `NOTOK`. Otherwise, `OK` is returned.

The `ll_log` routine is actually a simple wrapper around the `_ll_log` routine:

```
int     _ll_log (lp, event, ap)
LLog   *lp;
int     event;
va_list ap;
```

The parameters to this routine are:

`lp`: a pointer to a `LLog` structure;

`event`: the event type being logged; and,

`ap`: an argument pointer to a variable-length argument list as described in *varargs*(3).

It may be necessary to have multi-line log entries. In this case, the first line of the entry should be made with `ll_log`. The remaining lines should be made with the `ll_printf` routine:

```
int     ll_printf (lp, fmt, args ...)
LLog   *lp;
char   *fmt;
```

The parameters to this routine are:

`lp`: a pointer to a `LLog` structure; and,

`fmt/args ...`: an argument list to *printf*(3s).

As with `ll_log`, this routine returns either `OK` on success or `NOTOK` on error. Unlink `ll_log` however, `ll_printf` will ignore the setting of the `LLOGCLS` option. As such, when the last line of a multi-line entry has been made, the routine `ll_sync` should always be called to synchronize the log:

```
int     ll_sync (lp)
LLog   *lp;
```

The parameter to this routine is:

`lp`: a pointer to a `LLog` structure.

7.1.3 More About Making Log Entries

Although the `ll_log` routine has a basic functionality, programmers often prefer a slightly simpler interface. A few macros have been defined for this purpose.

The `SLOG` macro is the most commonly used:

```
SLOG (lp, event, what, args)
```

The parameters to this macro are:

`lp`: a pointer to a `LLog` structure;

`event`: the event type being logged;

`what`: some text associated with a system call error (use the manifest constant `NULLCP` if the entry is not associated with an error in a system call); and,

`args`: a parenthesized argument list for *printf*(3s).

The `SLOG` macro compares the event enabled for a log to the event being logged to see if `ll_log` should be called.

Since, the need for a `what` parameter is not common in many applications, the `LLOG` macro has been supplied. It is essentially the `SLOG` macro but with a value of `NULLCP` supplied for the `what` parameter of `ll_log`:

```
LLOG (lp, what, args)
```

Further, even though logging is contingent on an event type being enabled, a programmer may still wish that calls to logging package still be conditionally compiled. The `DLOG` macro has been supplied for this purpose. If the pre-processor symbol `DEBUG` is defined, then `DLOG` is equivalent to `LLOG` otherwise it compiles no code whatsoever:

```
DLOG (lp, what, args)
```

Finally, it may be useful to log PDUs (protocol data units), again under conditional compilation. The `PLOG` macro takes the address of a pretty-printer function generated by *pepy*(1) (see Section 6.5 on page 75 in *Volume Four*) along with a presentation element (as described in Chapter 5 in *Volume One*) and a brief textual title, and directs the pretty-printer to output to the log:

```
PLOG (lp, fnx, pe, text, rw)
```

The `rw` parameter is an integer saying wheter the PDU was read from the network (non-zero) or written to the network (zero-valued). As with the `DLOG` macro, if the `DDEBUG` symbol is not defined, then no code is generated.

7.1.4 Miscellaneous Routines

In order to support some of the more esoteric log capabilities, there are a few utility routines.

The routine `ll_preset` evaluates a *printf*(3s) argument list and returns a pointer to static buffer containing the result:

```
char   *ll_preset (fmt, args ...)
char   *fmt;
```

The routine `ll_check` determines if a log has exceeded its size, and if so, if a correction can be made:

```
int     ll_check (lp)
LLog   *lp;
```

This routine returns `OK` if the log is within its bounds.

7.2 Use of Logging in Programs

From the perspective of applications programmers, there are three kinds of styles for using the logging package, depending on the kind of program being written.

In all three cases, `LLog` structures are usually declared statically in the main module of a program and a pointer to the structure is made available for general use:

```
static LLog _pgm_log = {
    "myname.log", NULLCP, NULLCP,
    LLOG_FATAL | LLOG_EXCEPTIONS | LLOG_NOTICE,
    LLOG_FATAL,
    -1,
    LLOGCLS | LLOGCRT | LLOGZER,
```

```
        NOTOK
    };

    LLog *pgm_log = &_pgm_log;
```

Where the `myname` in "`myname.log`" is replaced with the name of the program.

Note that in all cases, in order to ensure consistent logging it is **critical** that the call to `isodetailor` be the first call made to *any* of the ISODE routines.

For static responders, two routines are called in the initialization code:

```
isodetailor (argv[0], 0);
ll_hdinit (pgm_log, argv[0]);
```

Later on, after argument parsing, if a debug option is enabled, then

```
ll_dbinit (pgm_log, argv[0]);
```

is called.

For dynamic responders, a similiar code sequence is used:

```
isodetailor (argv[0], 0);
if (debug = isatty (fileno (stderr)))
    ll_hdinit (pgm_log, argv[0]);
else
    ll_dbinit (pgm_log, argv[0]);
```

For user-interfaces, the code is simply:

```
isodetailor (argv[0], 1);
ll_hdinit (pgm_log, argv[0]);
```

which will ask the tailoing system to read both the standard tailoring file and a user-specific tailoring file and then to initialize the program log.

Part IV

Configuration

Chapter 8

The Transport Switch

As of this writing, the concept of ubiquitous OSI has not yet been realized. In particular, there continues to be disagreement on the transport/network protocols to be used to provide end-to-end service. As a result, interworking between sites in many cases is the *exception*, not the rule.

In order to facility communication, the ISODE has a powerful abstraction, the *Transport Switch*. The goal of the transport switch is to hide the complex underpinnings of the "real world" of OSI from the user of the transport service. In order to make effective use of this abstraction, it may be necessary to tailor the ISODE (using the *isotailor*(5) file described in Chapter 6).

To explain how to use the transport switch effectively, it is necessary to introduce some terminology.

8.1 Transport Stacks

A *TS-stack* refers to a combination of transport protocol and network service that is used to provide end-to-end transport service. The ISODE, depending on how you configure it at compile-time, supports any combination of these stacks:

Mneumoic	**TS-Stack**
`tcp`	RFC1006 over TCP/IP
`x25`	TP0 over X.25
`x25-84`	TP0 over X.25(1984)
`bridge`	TP0 over the TP0–bridge
`tp4`	TP4 over CLNP

Internally, the ISODE uses "typing" on all network addresses to one of these choices. From a practical perspective, usually only the `tcp`, `x25-84` and `x25` are of interest.

The run-time tailoring variable `ts_stacks` keeps track of the TS-stacks available on your host. This defaults to the TS-stacks that were configured at compile-time. However, if you are sharing executables between machines with different network attachments, you will want to change this. Generally, you compile-time configure the ISODE for all stacks available at your site. Then, for each host you install the executables on, you set the variable `ts_stacks` accordingly. So, suppose you have a host with both TCP and X.25 services, but that all the other hosts at your site have only TCP. In this case (assuming binary compatibility between the hosts at your site), you generate the ISODE on the host with both TCP and X.25 services configured. When you move the executables to the TCP only hosts, you add the line

```
ts_stacks: tcp
```

to the *isotailor* file on those hosts.

8.2 OSI Communities

An OSI *community* is a collection of hosts which share a common TS-stack along with basic connectivity. Simply put, a community consists of end-systems that all interwork with each other. In a perfect world, there would be but a single community. `realns`. But it isn't a perfect world, my favorite Rock group, *Bangles*, broke up in September of '89, and I am very upset about this. But that is another story.

There are several communities using OSI at present, the short list is:

Mneumoic	**Community**	**TS-stack**
`int-x25`	the International X.25	`x25`
`janet`	the JANET in the UK	`x25`
`internet`	the capital-I Internet	`tcp`
`localTCP`	the TCP loopback address	`tcp`
`IXI`	International X.25 Interconnect	`x25`

These are all termed *interim* communities as they do not use the real OSI network service. In order to facility communications between these communities (and others, e.g., private LANs), an Interim addressing scheme has been defined [Kil89b] (which is included in the ISODE documentation set, look in the directory *doc/interim/* in the source tree).

Thus, the first task when running the ISODE is to determine which communities your site belongs to. The `ts_communities` run-time tailoring variable keeps track of the communities which your host can (in)directly reach. This defaults to the four communities above, plus the `realNS` community. Whilst this is the most sensible default, this choice is probably wrong for most sites. So, if your host is connected to the International X.25 but not the JANET, you add the line

```
ts_communities: int-x25
```

to the *isotailor* file on that host.

Of course, if the host in question is connected not only to the International X.25 but also belongs to the capital-I Internet, then the line might read

```
ts_communities: int-x25 internet
```

instead. This raises an important question: if a host wishes to contact another host, and the two have multiple communities in common, what preference is given when connections are attempted? The answer is that the ordering of the communities in the `ts_communities` run-time tailoring variable tells the ISODE what the preference is. Thus, in the example above, if there were two hosts connected to both the International X.25 and the capital-I Internet, and one of the hosts wished to talk to the other, the International X.25 community would be tried first. If a connection could not be established, then the capital-I Internet community would be tried.

There are a few common site configurations. If the site has access to the capital-I Internet, then usually all hosts belong to the capital-I Internet community, e.g.,

```
ts_communities: internet
```

It is possible that one or more hosts at this site may have access to the International X.25, and exactly these hosts would have connectivity to this second community:

```
ts_communities: internet int-x25
```

8.2.1 Defining a new OSI community

Another common configuration is that a site has a single host that is connected to the International X.25. In addition, there is a private LAN, running the TCP/IP protocols, that all hosts at the site are connected to. In this case, you need to define a community name for your site. This is done by describing your community name as a macro in the *isomacros*(5) file and then enabling use of this community in the *isotailor*(5) file.

To describe your community, you create a macro of the form:

```
name    TELEX+value+stack+number+
```

where:

`name:` is the name of your community;

`value:` corresponds to the TELEX number at your site. (This is traditionally a three-digit international code followed by a national TELEX number.) The combination of `TELEX+value` defines an OSI address prefix which your site uniquely administers;

`stack:` is either `RFC-1006` or `X.25(80)` indicating the TS-stack used for the addresses; and,

`number:` by convention is a two-digit decimal number from 01 to 99 (this allows you to define up to 99 communities at your site).

Of course, there are other methods for generating unique OSI addresses, but this is the only method supported in the current implementation of the ISODE.

Defining a new TCP-based community

Consider the the case of an isolated LAN running the TCP/IP protocols.

First, edit the file *support/macros.local* in the source area and add a line line like this:

```
nott-ether TELEX+00738700+RFC-1006+01+
```

where `00738700` corresponds to your TELEX number. Next, type

```
# ./make macros
```

as the super-user. This will install a new version of the *isomacros*(5) file.

Second, in the *isotailor*(5) file, you would add this line:

```
ts_interim: nott-ether
```

which tells the ISODE about your new community. You would then add `nott-ether` to the beginning of the value for the run-time tailoring variable `ts_communities`. So, on the hosts with only TCP, you would have:

```
ts_interim:     nott-ether
ts_communities: nott-ether
```

and on the host which also had connectivity to the International X.25, you would have:

```
ts_interim:     nott-ether
ts_communities: nott-ether int-x25
```

Note that it is **critical** that the definition of `ts_interim` occur **before** the definition of `ts_communities`.

Defining a new X.25-based community

Consider the case of a site which has access to the International X.25 and belongs to a private X.25 network.

In this case, the macro definition might be:

```
psi-wan     TELEX+00738700+X.25(80)+02+
```

So, the tailoring variables would be

```
ts_interim: psi-wan
ts_communities: psi-wan
```

for those hosts which are only on the private X.25 network, whilst

```
ts_interim: psi-wan
ts_communities: psi-wan int-x25
```

would be used for hosts on both the International X.25 and the private X.25 network. Again, it is critical that the definition of `ts_interim` occur before the definition of `ts_communities`.

8.2.2 Heuristic Support

Finally, there are some cases when the ISODE must look at a simple string and derive an address. If the string encoding for presentation address defined in [Kil89a], then the macro defined earlier will help. But, if a simpler string is used, e.g.,

```
% ftam gonzo
```

or

```
% ftam 00000511160013
```

then it would be nice to have an intelligent default to use for the community associated with these. There are three run-time tailoring variables provided for this purpose:

if it looks like	**then use**
an OSI NSAP	`default_nsap_community`
a X.25 DTE	`default_x25_community`
a TCP address	`default_tcp_community`

To continue our two examples of private communities. for the private TCP/IP LAN, it would make sense to add

```
default_tcp_community: nott-ether
```

to the *isotailor* file on each host. For the private X.25–based community, it would make sense to add

```
default_x25_community: psi-wan
```

instead.

8.3 Transport-Service Bridges

There is one last question which must be considered: how can two hosts communicate if they do not have any communities in common? If there is a third host, which shares a community with the two other hosts, then a *TS-bridge* can be used to achieve connectivity on a per-connection basis.

8.3.1 Client Hosts

For each host, examine the value of the `ts_communities` run-time variable. For those communities which are not listed, but which you wish that host to be able to communicate with, you must add the definition of the appropriate TS-bridge to the `tsb_communities` run-time tailoring variable. A definition consists of two tokens, the name of the community that the TS-bridge can reach, and the transport address of the TS-bridge (using the string encoding defined in [Kil89a]), e.g.,

```
tsb_communities: internet Int-X25(80)=31344152401010+PID+03018000
```

This says that the capital-I Internet community can be reached by contacting the TS-bridge located on the International X.25 at the specified DTE and PID.

It is **critical** to observe that the TS-bridge must share a common community with the host containing this definition. Thus, a more instructive example is:

```
ts_communities: int-x25 internet
tsb_communities: internet Int-X25(80)=31344152401010+PID+03018000
```

which describes a host connected to the International X.25 that may wish to talk to hosts in the capital-I Internet community.

Similarly, one could imagine a host belonging to the capital-I Internet community using the following definitions:

```
ts_communities: internet int-x25
tsb_communities: int-x25 Internet=gonzo.twg.com+17004
```

8.3.2 Server Hosts

Typically, the entity responsible for a community also runs a TS-bridge which connects that community to the other Interim communities. Obviously, this host must support each of the Interim communities to be connected to the local community. The `tsb_default_address` run-time tailoring variable is used to define the transport address (usually containing multiple network addresses) where the TS-bridge listens.

So, to continue with the nott-ether example, the host on which the TS-bridge resides might have these definitions:

```
ts˙interim:      nott-ether
ts˙communities:      nott-ether int-x25
tsb˙default˙address: Nott-Ether=sheriff+17004“—Int-X25(80)=23426020017299+PID+03018000
```

The rule is simple, for each community named in the `ts_communities` run-time tailoring variable, a network address is present in the run-time tailoring variable `tsb_default_address`. By convention, for TCP-based addresses, TCP port 17004 is used, whilst for X.25–based addresses, PID 0301800 is used.

8.4 In Retrospect

What a mess! There's a lot to be said for the focused "one transport protocol, one network service" approach used by the Internet suite of protocols.

Bibliography

[CCI84a] Session Protocol Specification for Open Systems Interconnection (OSI) for CCITT Applications. International Telegraph and Telephone Consultative Committee, October 1984. Recommendation X.225.

[CCI84b] Session Service Definition for Open Systems Interconnection (OSI) for CCITT Applications. International Telegraph and Telephone Consultative Committee, October 1984. Recommendation X.224.

[CCI84c] Transport Service Definition for Open Systems Interconnection (OSI) for CCITT Applications. International Telegraph and Telephone Consultative Committee, October 1984. Recommendation X.214.

[ISO86] Information Processing Systems — Open Systems Interconnection — Transport Service Definition. International Organization for Standardization and International Electrotechnical Committee, 1986. International Standard 8072.

[ISO87a] 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.

[ISO87b] Information Processing Systems — Open Systems Interconnection — Basic Connection Oriented Session Protocol Specification. International Organization for Standardization and International

Electrotechnical Committee, August 1987. International Standard 8327.

[ISO87c] Information Processing Systems — Open Systems Interconnection — Basic Connection Oriented Session Service Definition — Addendum 2: Incorporation of Unlimited User Data. International Organization for Standardization and International Electrotechnical Committee, August 1987. Draft Addendum 8326/DAD 2.

[ISO87d] Information Processing Systems — Open Systems Interconnection — Basic Connection Oriented Session Protocol Specification — Addendum 2: Incorporation of Unlimited User Data. International Organization for Standardization and International Electrotechnical Committee, August 1987. Draft Addendum 8327/DAD 2.

[ISO87e] Information Processing Systems — Open Systems Interconnection — Basic Connection Oriented Session Service Definition. International Organization for Standardization and International Electrotechnical Committee, August 1987. International Standard 8326.

[ISO88a] Information Processing Systems — Open Systems Interconnection — Connection Oriented Presentation Service Definition. International Organization for Standardization and International Electrotechnical Committee, April 1988. Final Text of Draft International Standard 8822.

[ISO88b] Information Processing Systems — Open Systems Interconnection — Connection Oriented Presentation Protocol Specification. International Organization for Standardization and International Electrotechnical Committee, April 1988. Final Text of Draft International Standard 8823.

[Kil89a] Stephen E. Kille. A string encoding of Presentation Address. Research Note RN/89/14, Department of Computer Science, University College London, February 1989.

[Kil89b] Stephen E. Kille. An interim approach to use of Network Addresses. Research Note RN/89/13, 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.

[Pad85] Michael A. Padlipksy. Gateways, Architectures, and Heffalumps. In *The Elements of Networking Style and Other Essays and Animadversions on the Art of Intercomputer Networking*, chapter 10, pages 167–176. Prentice-hall, Englewood Cliffs, New Jersey, 1985. Also available as Internet Request for Comments 875.

[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.

[RC87] Marshall T. Rose and Dwight E. Cass. ISO Transport Services on top of the TCP. Request for Comments 1006, DDN Network Information Center, SRI International, May 1987.

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

Index